Curs practic de Java

Cristian Frăsinaru

Cuprins

Capitolul 1

Introducere în Java

1.1 Ce este Java ?

Java este o tehnologie inovatoare lansată de compania Sun Microsystems în 1995, care a avut un impact remarcabil asupra întregii comunităţi a dezvoltatorilor de software, impunându-se prin calităţi deosebite cum ar fi simplitate, robusteţe şi nu în ultimul rând portabilitate. Denumită iniţial *OAK*, tehnologia Java este formată dintr-un limbaj de programare de nivel înalt pe baza căruia sunt construite o serie de platforme destinate implementării de aplicaţii pentru toate segmentele industriei software.

1.1.1 Limbajul de programare Java

Inainte de a prezenta în detaliu aspectele tehnice ale limbajului Java, să amintim caracteristicile sale principale, care l-au transformat într-un interval de timp atât de scurt într-una din cele mai pupulare opţiuni pentru dezvoltarea de aplicaţii, indiferent de domeniu sau de complexitatea lor.

- **Simplitate** - elimină supraîncărcarea operatorilor, moştenirea multiplă şi toate "facilităţile" ce pot provoca scrierea unui cod confuz.
- **Uşurinţă** în crearea de aplicaţii complexe ce folosesc programarea în reţea, fire de execuţie, interfaţă grafică, baze de date, etc.
- **Robusteţe** - elimină sursele frecvente de erori ce apar în programare prin renunţarea la pointeri, administrarea automată a memoriei şi elim-

inarea pierderilor de memorie printr-o procedură de colectare a obiectelor care nu mai sunt referite, ce rulează în fundal ("garbage collector").

- **Complet orientat pe obiecte** - elimină complet stilul de programare procedural.
- **Securitate** - este un limbaj de programare foarte sigur, furnizând mecanisme stricte de securitate a programelor concretizate prin: verificarea dinamică a codului pentru detectarea secvenţelor periculoase, impunerea unor reguli stricte pentru rularea proceselor la distanţă, etc.
- **Neutralitate arhitecturală** - comportamentul unei aplicaţii Java nu depinde de arhitectura fizică a maşinii pe care rulează.
- **Portabililtate** - Java este un limbaj independent de platforma de lucru, aceeaşi aplicaţie rulând fără nici o modificare şi fără a necesita recompilarea ei pe sisteme de operare diferite cum ar fi Windows, Linux, Mac OS, Solaris, etc. lucru care aduce economii substanţiale firmelor dezvoltatoare de aplicaţii.
- Este **compilat şi interpretat**, aceasta fiind soluţia eficientă pentru obţinerea portabilităţii.
- **Performanţă** - deşi mai lent decât limbajele de programare care generează executabile native pentru o anumită platformă de lucru, compilatorul Java asigură o performanţă ridicată a codului de octeţi, astfel încât viteza de lucru puţin mai scăzută nu va fi un impediment în dezvoltarea de aplicaţii oricât de complexe, inclusiv grafică 3D, animaţie, etc.
- Este **modelat după C şi C++**, trecerea de la C, C++ la Java făcându-se foarte uşor.

1.1.2 Platforme de lucru Java

Limbajul de programare Java a fost folosit la dezvoltarea unor tehnologii dedicate rezolvării unor probleme din cele mai diverse domenii. Aceste tehnologii au fost grupate în aşa numitele *platforme de lucru*, ce reprezintă seturi de librării scrise în limbajul Java, precum şi diverse programe utilitare, folosite pentru dezvoltarea de aplicaţii sau componente destinate unei anume categorii de utilizatori.

- **J2SE** (Standard Edition)
  Este platforma standard de lucru ce oferă suport pentru crearea de aplicaţii independente şi appleturi.

  De asemenea, aici este inclusă şi tehnologia **Java Web Start** ce furnizează o modalitate extrem de facilă pentru lansarea şi instalarea locală a programelor scrise în Java direct de pe Web, oferind cea mai comodă soluţie pentru distribuţia şi actualizarea aplicaţiilor Java.

- **J2ME** (Micro Edition)
  Folosind Java, programarea dispozitivelor mobile este extrem de simplă, platforma de lucru J2ME oferind suportul necesar scrierii de programe dedicate acestui scop.

- **J2EE** (Enterprise Edition)
  Această platformă oferă API-ul necesar dezvoltării de aplicaţii complexe, formate din componente ce trebuie să ruleze în sisteme eterogene, cu informaţiile memorate în baze de date distribuite, etc.

  Tot aici găsim şi suportul necesar pentru crearea de **aplicaţii** şi **servicii Web**, bazate pe componente cum ar fi servleturi, pagini JSP, etc.

Toate distribuţiile Java sunt oferite **gratuit** şi pot fi descărcate de pe Internet de la adresa "http://java.sun.com".

In continuare, vom folosi termenul J2SDK pentru a ne referi la distribuţia standard J2SE 1.5 SDK (Tiger).

1.1.3 Java: un limbaj compilat şi interpretat

In funcţie de modul de execuţie a aplicaţiilor, limbajele de programare se împart în două categorii:

- **Interpretate**: instrucţiunile sunt citite linie cu linie de un program numit *interpretor* şi traduse în instrucţiuni maşină. Avantajul acestei soluţii este simplitatea şi faptul că fiind interpretată direct sursa programului obţinem portabilitatea. Dezavantajul evident este viteza de execuţie redusă. Probabil cel mai cunoscute limbaj interpretat este limbajul Basic.

- **Compilate**: codul sursă al programelor este transformat de *compilator* într-un cod ce poate fi executat direct de procesor, numit *cod*

maşină. Avantajul este execuţia extrem de rapidă, dezavantajul fiind lipsa portabilităţii, codul compilat într-un format de nivel scăzut nu poate fi rulat decât pe platforma de lucru pe care a fost compilat.

Limbajul Java combină soluţiile amintite mai sus, programele Java fiind atât interpretate cât şi compilate. Aşadar vom avea la dispoziţie un compilator responsabil cu transformarea surselor programului în aşa numitul *cod de octeţi*, precum şi un interpretor ce va executa respectivul cod de octeţi.

Codul de octeţi este diferit de codul maşină. Codul maşină este reprezentat de o succesiune de instrucţiuni specifice unui anumit procesor şi unei anumite platforme de lucru reprezentate în format binar astfel încât să poată fi executate fără a mai necesita nici o prelucrare.

Codurile de octeţi sunt seturi de instrucţiuni care seamănă cu codul scris în limbaj de asamblare şi sunt generate de compilator independent de mediul de lucru. In timp ce codul maşină este executat direct de către procesor şi poate fi folosit numai pe platforma pe care a fost creat, codul de octeţi este interpretat de mediul Java şi de aceea poate fi rulat pe orice platformă pe care este instalată mediul de execuţie Java.

Prin *maşina virtuală Java (JVM)* vom înţelege mediul de execuţie al aplicaţiilor Java. Pentru ca un cod de octeţi să poată fi executat pe un anumit calculator, pe acesta trebuie să fie instalată o maşină virtuală Java. Acest lucru este realizat automat de către distribuţia J2SDK.

1.2 Primul program

Crearea oricărei aplicaţii Java presupune efectuarea următorilor paşi:

1. **Scriererea codului sursă**

```
class FirstApp {
  public static void main( String args[]) {
    System.out.println("Hello world!");
  }
}
```

Toate aplicaţiile Java conţin o clasă principală(primară) în care trebuie să se gasească metoda `main`. Clasele aplicaţiei se pot gasi fie într-un singur fişier, fie în mai multe.

2. Salvarea fişierelor sursă

Se va face în fişiere care au obligatoriu extensia `java`, nici o altă extensie nefiind acceptată. Este recomandat ca fişierul care conţine codul sursă al clasei primare să aibă acelaşi nume cu cel al clasei, deşi acest lucru nu este obligatoriu. Să presupunem că am salvat exemplul de mai sus în fişierul `C:\intro\FirstApp.java`.

3. Compilarea aplicaţiei

Pentru compilare vom folosi compilatorul **javac** din distribuţia J2SDK. Apelul compilatorului se face pentru fişierul ce conţine clasa principală a aplicaţiei sau pentru orice fişier/fişiere cu extensia `java`. Compilatorul creează câte un fişier separat pentru fiecare clasă a programului. Acestea au extensia `.class` şi implicit sunt plasate în acelaşi director cu fişierele sursă.

```
javac FirstApp.java
```

In cazul în care compilarea a reuşit va fi generat fişierul `FirstApp.class`.

4. Rularea aplicaţiei

Se face cu interpretorul **java**, apelat pentru unitatea de compilare corespunzătoare clasei principale. Deoarece interpretorul are ca argument de intrare numele clasei principale şi nu numele unui fişier, ne vom poziţiona în directorul ce conţine fişierul `FirstApp.class` şi vom apela interpretorul astfel:

```
java FirstApp
```

Rularea unei aplicaţii care nu foloseşte interfaţă grafică, se va face într-o fereastră sistem.

Atenţie
Un apel de genul `java c:\intro\FirstApp.class` este greşit!

1.3 Structura lexicală a limbajului Java

1.3.1 Setul de caractere

Limbajului Java lucrează în mod nativ folosind setul de caractere Unicode. Acesta este un standard internaţional care înlocuieşte vechiul set de caractere ASCII şi care foloseşte pentru reprezentarea caracterelor 2 octeţi, ceea ce înseamnă că se pot reprezenta 65536 de semne, spre deosebire de ASCII, unde era posibilă reprezentarea a doar 256 de caractere. Primele 256 caractere Unicode corespund celor ASCII, referirea la celelalte făcându-se prin `\uxxxx`, unde `xxxx` reprezintă codul caracterului.

O altă caracteristică a setului de caractere Unicode este faptul că întreg intervalul de reprezentare a simbolurilor este divizat în subintervale numite **blocuri**, câteva exemple de blocuri fiind: Basic Latin, Greek, Arabic, Gothic, Currency, Mathematical, Arrows, Musical, etc.

Mai jos sunt oferite câteva exemple de caractere Unicode.

- `\u0030 - \u0039` : cifre ISO-Latin 0 - 9
- `\u0660 - \u0669` : cifre arabic-indic 0 - 9
- `\u03B1 - \u03C9` : simboluri greceşti $\alpha - \omega$
- `\u2200 - \u22FF` : simboluri matematice ($\forall, \exists, \emptyset$, etc.)
- `\u4e00 - \u9fff` : litere din alfabetul Han (Chinez, Japonez, Coreean)

Mai multe informaţii legate de reprezentarea Unicode pot fi obţinute la adresa "http://www.unicode.org".

1.3.2 Cuvinte cheie

Cuvintele rezervate în Java sunt, cu câteva excepţii, cele din C++ şi au fost enumerate în tabelul de mai jos. Acestea nu pot fi folosite ca nume de clase,

interfeţe, variabile sau metode. `true, false, null` nu sunt cuvinte cheie, dar nu pot fi nici ele folosite ca nume în aplicaţii. Cuvintele marcate prin $*$ sunt rezervate, dar nu sunt folosite.

abstract	double	int	strictfp
boolean	else	interface	super
break	extends	long	switch
byte	final	native	synchronized
case	finally	new	this
catch	float	package	throw
char	for	private	throws
class	goto*	protected	transient
const*	if	public	try
continue	implements	return	void
default	import	short	volatile
do	instanceof	static	while

Incepând cu versiunea 1.5, mai există şi cuvântul cheie `enum`.

1.3.3 Identificatori

Sunt secvenţe nelimitate de litere şi cifre Unicode, începând cu o literă. După cum am mai spus, identificatorii nu au voie să fie identici cu cuvintele rezervate.

1.3.4 Literali

Literalii pot fi de următoarele tipuri:

- **Intregi**
  Sunt acceptate 3 baze de numeraţie : baza 10, baza 16 (încep cu caracterele 0x) şi baza 8 (încep cu cifra 0) şi pot fi de două tipuri:
  - **normali** - se reprezintă pe 4 octeţi (32 biţi)
  - **lungi** - se reprezintă pe 8 octeţi (64 biţi) şi se termină cu caracterul L (sau l).

- **Flotanţi**
  Pentru ca un literal să fie considerat flotant el trebuie să aibă cel puţin o zecimală după virgulă, să fie în notaţie exponenţială sau să aibă sufixul F sau f pentru valorile normale - reprezentate pe 32 biţi, respectiv D sau d pentru valorile duble - reprezentate pe 64 biţi.
  Exemple: $1.0, 2e2, 3f, 4D$.

- **Logici**
  Sunt reprezentaţi de `true` - valoarea logică de adevăr, respectiv `false` - valoarea logică de fals.

Atenţie

Spre deosebire de C++, literalii întregi 1 şi 0 nu mai au semnificaţia de adevărat, respectiv fals.

- **Caracter**
  Un literal de tip caracter este utilizat pentru a exprima caracterele codului Unicode. Reprezentarea se face fie folosind o literă, fie o secvenţă *escape* scrisă între apostrofuri. Secvenţele escape permit specificarea caracterelor care nu au reprezentare grafică şi reprezentarea unor caractere speciale precum backslash, apostrof, etc. Secvenţele escape predefinite în Java sunt:
  - `'\b'` : Backspace (BS)
  - `'\t'` : Tab orizontal (HT)
  - `'\n'` : Linie nouă (LF)
  - `'\f'` : Pagină nouă (FF)
  - `'\r'` : Inceput de rând (CR)
  - `'\"'` : Ghilimele
  - `'\''` : Apostrof
  - `'\\'` : Backslash

- **Şiruri de caractere**
  Un literal şir de caractere este format din zero sau mai multe caractere între ghilimele. Caracterele care formează şirul pot fi caractere grafice sau secvenţe escape.
  Dacă şirul este prea lung el poate fi scris ca o concatenare de subşiruri de dimensiune mai mică, concatenarea şirurilor realizându-se cu operatorul +, ca în exemplul: `"Ana " + " are " + " mere "`. Sirul vid este `""`.

  După cum vom vedea, orice şir este de fapt o instanţă a clasei `String`, definită în pachetul `java.lang`.

1.3.5 Separatori

Un separator este un caracter care indică sfârşitul unei unităţi lexicale şi ınceputul alteia. In Java separatorii sunt următorii: `( )  [ ]  ; , .` . Instrucţiunile unui program se separă cu punct şi virgulă.

1.3.6 Operatori

Operatorii Java sunt, cu mici deosebiri, cei din C++:

- atribuirea: `=`
- operatori matematici: `+, -, *, /, %, ++, --` .
  Este permisă notaţia prescurtată de forma `lval op= rval`: `x += 2 n -= 3`
  Există operatori pentru autoincrementare şi autodecrementare (post şi pre): `x++, ++x, n--, --n`
  Evaluarea expresiilor logice se face prin metoda *scurtcircuitului*: evaluarea se opreşte în momentul în care valoarea de adevăr a expresiei este sigur determinată.
- operatori logici: `&&(and), ||(or), !(not)`
- operatori relaţionali: `<, <=, >, <=, ==, !=`
- operatori pe biţi: `&(and), |(or), ^ (xor), ~ (not)`
- operatori de translaţie: `<<, >>, >>> (shift la dreapta fără semn)`

- operatorul *if-else*: `expresie-logica ? val-true : val-false`
- operatorul , (virgulă) folosit pentru evaluarea secvenţială a operaţiilor: `int x=0, y=1, z=2;`
- operatorul + pentru concatenarea şirurilor:

```
String s1="Ana";
String s2="mere";
int x=10;
System.out.println(s1 + " are " + x + " " + s2);
```

- operatori pentru conversii (*cast*) : `(tip-de-data)`

```
int a = (int)'a';
char c = (char)96;
int i = 200;
long l = (long)i; //widening conversion
long l2 = (long)200;
int i2 = (int)l2; //narrowing conversion
```

1.3.7 Comentarii

In Java există trei feluri de comentarii:

- Comentarii pe mai multe linii, închise între /* şi */.
- Comentarii pe mai multe linii care ţin de documentaţie, închise între /** şi */. Textul dintre cele două secvenţe este automat mutat în documentaţia aplicaţiei de către generatorul automat de documentaţie **javadoc**.
- Comentarii pe o singură linie, care incep cu `//`.

Observaţii:

- Nu putem scrie comentarii în interiorul altor comentarii.
- Nu putem introduce comentarii în interiorul literalilor caracter sau şir de caractere.
- Secvenţele `/*` şi `*/` pot să apară pe o linie după secvenţa `//` dar îşi pierd semnificaţia. La fel se întamplă cu secvenţa `//` în comentarii care incep cu `/*` sau `*/`.

1.4 Tipuri de date şi variabile

1.4.1 Tipuri de date

In Java tipurile de date se impart în două categorii: **tipuri primitive** şi **tipuri referinţă**. Java porneşte de la premiza că "orice este un obiect", prin urmare tipurile de date ar trebui să fie de fapt definite de clase şi toate variabilele ar trebui să memoreze instanţe ale acestor clase (obiecte). In principiu acest lucru este adevărat, însă, pentru usurinţa programării, mai există şi aşa numitele tipurile primitive de date, care sunt cele uzuale :

- **aritmetice**
  - **întregi**: `byte` (1 octet), `short` (2), `int` (4), `long` (8)
  - **reale**: `float` (4 octeti), `double` (8)
- **caracter**: `char` (2 octeţi)
- **logic**: `boolean` (`true` şi `false`)

In alte limbaje de programare formatul şi dimensiunea tipurilor primitive de date pot depinde de platforma pe care rulează programul. In Java acest lucru nu mai este valabil, orice dependenţă de o anumită platformă specifică fiind eliminată.

Vectorii, clasele şi interfeţele sunt tipuri referinţă. Valoarea unei variabile de acest tip este, spre deosebire de tipurile primitive, o referinţă (adresă de memorie) către valoarea sau mulţimea de valori reprezentată de variabila respectivă.

Există trei tipuri de date din limbajul C care nu sunt suportate de limbajul Java. Acestea sunt: `pointer`, `struct` şi `union`. Pointerii au fost eliminaţi din cauză că erau o sursă constantă de erori, locul lor fiind luat de tipul referinţă, iar `struct` şi `union` nu îşi mai au rostul atât timp cât tipurile compuse de date sunt formate în Java prin intermediul claselor.

1.4.2 Variabile

Variabilele pot fi de tip primitiv sau referinţe la obiecte (tip referinţă). Indiferent de tipul lor, pentru a putea fi folosite variabilele trebuie declarate şi, eventual, iniţializate.

- Declararea variabilelor: `Tip numeVariabila;`
- Iniţializarea variabilelor: `Tip numeVariabila = valoare;`
- Declararea constantelor: `final Tip numeVariabila;`

Evident, există posibilitatea de a declara şi iniţializa mai multe variabile sau constante de acelaşi tip într-o singură instrucţiune astfel:

```
Tip variabila1[=valoare1], variabila2[=valoare2],...;
```

Convenţia de numire a variabilelor în Java include, printre altele, următoarele criterii:

- variabilele finale (constante) se scriu cu majuscule;
- variabilele care nu sunt constante se scriu astfel: prima literă mică iar dacă numele variabilei este format din mai mulţi atomi lexicali, atunci primele litere ale celorlalţi atomi se scriu cu majuscule.

Exemple:

```
final double PI = 3.14;
final int MINIM=0, MAXIM = 10;
int valoare = 100;
char c1='j', c2='a', c3='v', c4='a';
long numarElemente = 12345678L;
String bauturaMeaPreferata = "apa";
```

In funcţie de locul în care sunt declarate variabilele se împart în următoatele categorii:

a. Variabile membre, declarate în interiorul unei clase, vizibile pentru toate metodele clasei respective cât şi pentru alte clase în funcţie de nivelul lor de acces (vezi "Declararea variabilelor membre").

b. Parametri metodelor, vizibili doar în metoda respectivă.

c. Variabile locale, declarate într-o metodă, vizibile doar în metoda respectivă.

d. Variabile locale, declarate într-un bloc de cod, vizibile doar în blocul respectiv.

e. Parametrii de la tratarea excepţiilor (vezi "Tratarea excepţiilor").

```
class Exemplu {
  //Fiecare variabila corespunde situatiei data de numele ei
  //din enumerarea de mai sus
  int a;
  public void metoda(int b) {
    a = b;
    int c = 10;
    for(int d=0; d < 10; d++) {
      c --;
    }
    try {
      a = b/c;
    } catch(ArithmeticException e) {
      System.err.println(e.getMessage());
    }
  }
}
```

Observatii:

- Variabilele declarate într-un `for`, rămân locale corpului ciclului:

```
for(int i=0; i<100; i++) {
  //domeniul de vizibilitate al lui i
}
i = 101;//incorect
```

- Nu este permisă ascunderea unei variabile:

```
int x=1;
{
  int x=2; //incorect
}
```

1.5 Controlul execuţiei

Instrucţiunile Java pentru controlul execuţiei sunt foarte asemănătoare celor din limbajul C şi pot fi împărţite în următoarele categorii:

- Instrucţiuni de decizie: `if-else, switch-case`
- Instrucţiuni de salt: `for, while, do-while`
- Instrucţiuni pentru tratarea excepţiilor: `try-catch-finally, throw`
- Alte instrucţiuni: `break, continue, return,` *label:*

1.5.1 Instrucţiuni de decizie

if-else

```
if (expresie-logica) {
  ...
}

if (expresie-logica) {
   ...
} else {
   ...
}
```

switch-case

```
switch (variabila) {
  case valoare1:
     ...
    break;
  case valoare2:
```

```
    ...
    break;
  ...
  default:
    ...
}
```

Variabilele care pot fi testate folosind instrucţiunea `switch` nu pot fi decât de tipuri primitive.

1.5.2 Instrucţiuni de salt

for

```
for(initializare; expresie-logica; pas-iteratie) {
  //Corpul buclei
}

for(int i=0, j=100 ; i < 100 && j > 0; i++, j--) {
  ...
}
```

Atât la iniţializare cât şi în pasul de iteraţie pot fi mai multe instrucţiuni despărţite prin virgulă.

while

```
while (expresie-logica) {
  ...
}
```

do-while

```
do {
  ...
}
while (expresie-logica);
```

1.5.3 Instrucţiuni pentru tratarea excepţiilor

Instrucţiunile pentru tratarea excepţiilor sunt `try-catch-finally`, respectiv `throw` şi vor fi tratate în capitolul "Excepţii".

1.5.4 Alte instrucţiuni

- `break`: părăseşte forţat corpul unei structuri repetitive.
- `continue`: termina forţat iteraţia curentă a unui ciclu şi trece la următoarea iteraţie.
- `return [valoare]`: termină o metodă şi, eventual, returnează o valorare.
- *numeEticheta:* : Defineşte o etichetă.

Deşi în Java nu există `goto`, se pot defini totuşi etichete folosite în expresii de genul: `break numeEticheata` sau `continue numeEticheta`, utile pentru a controla punctul de ieşire dintr-o structură repetitivă, ca înexemplul de mai jos:

```
i=0;
eticheta:
while (i < 10) {
  System.out.println("i="+i);
  j=0;
  while (j < 10) {
    j++;
    if (j==5) continue eticheta;
    if (j==7) break eticheta;
    System.out.println("j="+j);
  }
  i++;
}
```

1.6 Vectori

1.6.1 Crearea unui vector

Crearea unui vector presupune realizarea următoarelor etape:

- **Declararea** vectorului - Pentru a putea utiliza un vector trebuie, înainte de toate, sa-l declarăm. Acest lucru se face prin expresii de forma:

```
Tip[] numeVector; sau
Tip numeVector[];
```

ca în exemplele de mai jos:

```
int[] intregi;
String adrese[];
```

- **Instanţierea**

Declararea unui vector nu implică şi alocarea memoriei necesare pentru reţinerea elementelor. Operaţiunea de alocare a memoriei, numită şi instanţierea vectorului, se realizează întotdeauna prin intermediul operatorului `new`. Instanţierea unui vector se va face printr-o expresie de genul:

```
numeVector = new Tip[nrElemente];
```

unde *nrElemente* reprezintă numărul maxim de elemente pe care le poate avea vectorul. In urma instanţierii vor fi alocaţi: $nrElemente * dimensiune(Tip)$ octeţi necesari memorării elementelor din vector, unde prin $dimensiune(Tip)$ am notat numărul de octeţi pe care se reprezintă tipul respectiv.

```
v = new int[10];
//aloca spatiu pentru 10 intregi: 40 octeti

c = new char[10];
//aloca spatiu pentru 10 caractere: 20 octeti
```

Declararea şi instanţierea unui vector pot fi făcute simultan astfel:

```
Tip[] numeVector = new Tip[nrElemente];
```

- **Iniţializarea** (opţional) După declararea unui vector, acesta poate fi iniţializat, adică elementele sale pot primi nişte valori iniţiale, evident dacă este cazul pentru aşa ceva. In acest caz instanţierea nu mai trebuie facută explicit, alocarea memoriei făcându-se automat în funcţie de numă rul de elemente cu care se iniţializează vectorul.

```
String culori[] = {"Rosu", "Galben", "Verde"};
int []factorial = {1, 1, 2, 6, 24, 120};
```

Primul indice al unui vector este 0, deci poziţiile unui vector cu n elemente vor fi cuprinse între 0 şi $n-1$. Nu sunt permise construcţii de genul `Tip numeVector[nrElemente]`, alocarea memoriei făcându-se doar prin intermediul opearatorului `new`.

```
int v[10];              //ilegal
int v[] = new int[10];  //corect
```

1.6.2 Tablouri multidimensionale

In Java tablourile multidimensionale sunt de fapt vectori de vectori. De exemplu, crearea şi instanţierea unei matrici vor fi realizate astfel:

```
Tip matrice[][] = new Tip[nrLinii][nrColoane];
```

`matrice[i]` este linia i a matricii şi reprezintă un vector cu $nrColoane$ elemente iar `matrice[i][j]` este elementul de pe linia i şi coloana j.

1.6.3 Dimensiunea unui vector

Cu ajutorul variabilei **length** se poate afla numărul de elemente al unui vector.

```
int []a = new int[5];
// a.length are valoarea 5

int m[][] = new int[5][10];
// m[0].length are valoarea 10
```

Pentru a înţelege modalitatea de folosire a lui `length` trebuie menţionat că fiecare vector este de fapt o instanţă a unei clase iar `length` este o variabilă publică a acelei clase, în care este reţinut numărul maxim de elemente al vectorului.

1.6.4 Copierea vectorilor

Copierea elementelor unui vector a într-un alt vector b se poate face, fie element cu element, fie cu ajutorul metodei `System.arraycopy`, ca în exemplele de mai jos. După cum vom vedea, o atribuire de genul $b = a$ are altă semnificaţie decât copierea elementelor lui a în b şi nu poate fi folosită în acest scop.

```
int a[] = {1, 2, 3, 4};
int b[] = new int[4];

// Varianta 1
for(int i=0; i<a.length; i++)
  b[i] = a[i];

// Varianta 2
System.arraycopy(a, 0, b, 0, a.length);

// Nu are efectul dorit
b = a;
```

1.6.5 Sortarea vectorilor - clasa `Arrays`

In Java s-a pus un accent deosebit pe implementarea unor structuri de date şi algoritmi care să simplifice proceseul de crearea a unui algoritm, programatorul trebuind să se concentreze pe aspectele specifice problemei abordate. Clasa `java.util.Arrays` oferă diverse metode foarte utile în lucrul cu vectori cum ar fi:

- **sort** - sortează ascendent un vector, folosind un algoritm de tip *QuickSort* performant, de complexitate $O(n\,log(n))$.

  ```
  int v[]={3, 1, 4, 2};
  java.util.Arrays.sort(v);
  // Sorteaza vectorul v
  // Acesta va deveni {1, 2, 3, 4}
  ```

- **binarySearch** - căutarea binară a unei anumite valori într-un vector sortat;

- **equals** - testarea egalităţii valorilor a doi vectori (au aceleaşi număr de elemente şi pentru fiecare indice valorile corespunzătoare din cei doi vectori sunt egale)
- **fill** - atribuie fiecărui element din vector o valoare specificată.

1.6.6 Vectori cu dimensiune variabilă şi eterogeni

Implementări ale vectorilor cu număr variabil de elemente sunt oferite de clase specializate cum ar fi `Vector` sau `ArrayList` din pachetul `java.util`. Astfel de obiecte descriu vectori eterogeni, ale căror elemente au tipul `Object`, şi vor fi studiaţi în capitolul "Colecţii".

1.7 Şiruri de caractere

In Java, un şir de caractere poate fi reprezentat printr-un vector format din elemente de tip `char`, un obiect de tip **String** sau un obiect de tip **StringBuffer**.

Dacă un şir de caractere este constant (nu se doreşte schimbarea conţinutului să pe parcursul execuţiei programului) atunci el va fi declarat de tipul `String`, altfel va fi declarat de tip `StringBuffer`. Diferenţa principală între aceste clase este că `StringBuffer` pune la dispoziţie metode pentru modificarea conţinutului şirului, cum ar fi: `append, insert, delete, reverse`.
Uzual, cea mai folosită modalitate de a lucra cu şiruri este prin intermediul clasei `String`, care are şi unele particularităţi faţă de restul claselor menite să simplifice cât mai mult folosirea şirurilor de caractere. Clasa `StringBuffer` va fi utilizată predominant în aplicaţii dedicate procesării textelor cum ar fi editoarele de texte.

Exemple echivalente de declarare a unui şir:

```
String s = "abc";
String s = new String("abc");
char data[] = {'a', 'b', 'c'};
String s = new String(data);
```

Observaţi prima variantă de declarare a şirului s din exemplul de mai sus - de altfel, cea mai folosită - care prezintă o particularitate a clasei `String` faţa de restul claselor Java referitoare la instanţierea obiectelor sale.

Concatenarea şirurilor de caractere se face prin intermediul operatorului + sau, în cazul şirurilor de tip `StringBuffer`, folosind metoda `append`.

```
String s1 = "abc" + "xyz";
String s2 = "123";
String s3 = s1 + s2;
```

In Java, operatorul de concatenare + este extrem de flexibil, în sensul că permite concatenarea şirurilor cu obiecte de orice tip care au o reprezentare de tip şir de caractere. Mai jos, sunt câteva exemple:

```
System.out.print("Vectorul v are" + v.length + " elemente");
String x = "a" + 1 + "b"
```

Pentru a lămuri puţin lucrurile, ceea ce execută compilatorul atunci când întâlneşte o secvenţă de genul `String x = "a" + 1 + "b"` este:

```
String x = new StringBuffer().append("a").append(1).
 append("b").toString()
```

Atenţie însă la ordinea de efectuare a operaţiilor. Şirul `s=1+2+"a"+1+2` va avea valoarea `"3a12"`, primul + fiind operatorul matematic de adunare iar al doilea +, cel de concatenare a şirurilor.

1.8 Folosirea argumentelor de la linia de comandă

1.8.1 Transmiterea argumentelor

O aplicaţie Java poate primi oricâte argumente de la linia de comanda în momentul lansării ei. Aceste argumente sunt utile pentru a permite utilizatorului să specifice diverse opţiuni legate de funcţionarea aplicaţiei sau să furnizeze anumite date iniţiale programului.

Atenţie

Programele care folosesc argumente de la linia de comandă nu sunt 100% pure Java, deoarece unele sisteme de operare, cum ar fi Mac OS, nu au în mod normal linie de comandă.

Argumentele de la linia de comandă sunt introduse la lansarea unei aplicaţii, fiind specificate după numele aplicaţiei şi separate prin spaţiu. De exemplu, să presupunem că aplicaţia `Sortare` ordonează lexicografic (alfabetic) liniile unui fişier şi primeşte ca argument de intrare numele fişierului pe care să îl sorteze. Pentru a ordona fişierul `"persoane.txt"`, aplicaţia va fi lansată astfel:

```
java Sortare persoane.txt
```

Aşadar, formatul general pentru lansarea unei aplicaţii care primeşte argumente de la linia de comandă este:

```
java NumeAplicatie [arg0 arg1 . . . argn]
```

In cazul în care sunt mai multe, argumentele trebuie separate prin spaţii iar dacă unul dintre argumente conţine spaţii, atunci el trebuie pus între ghilimele. Evident, o aplicaţie poate să nu primească nici un argument sau poate să ignore argumentele primite de la linia de comandă.

1.8.2 Primirea argumentelor

In momentul lansării unei aplicaţii interpretorul parcurge linia de comandă cu care a fost lansată aplicaţtia şi, în cazul în care există, transmite programului argumentele specificate sub forma unui vector de şiruri. Acesta este primit de aplicaţie ca parametru al metodei `main`. Reamintim că formatul metodei `main` din clasa principală este:

```
public static void main (String args[])
```

Vectorul *args* primit ca parametru de metoda `main` va conţine toate argumentele transmise programului de la linia de comandă.
In cazul apelului `java Sortare persoane.txt` vectorul *args* va conţine un singur element pe prima să poziţie: `args[0]="persoane.txt"`.

Vectoru *args* este instanţiat cu un număr de elemente egal cu numărul argumentelor primite de la linia de comandă. Aşadar, pentru a afla numărul de argumente primite de program este suficient să aflăm dimensiunea vectorului `args` prin intermediul atributului `length`:

```
  public static void main (String args[]) {
    int numarArgumente = args.length ;
  }
```

In cazul în care aplicaţia presupune existenţa unor argumente de la linia de comandă, însă acestea nu au fost transmise programului la lansarea sa, vor apărea excepţii (erori) de tipul `ArrayIndexOutOfBoundsException`. Tratarea acestor excepţii este prezentată în capitolul "Excepţii".
Din acest motiv, este necesar să testăm dacă programul a primit argumentele de la linia de comandă necesare pentru funcţionarea sa şi, în caz contrar, să afişeze un mesaj de avertizare sau să folosească nişte valori implicite, ca în exemplul de mai jos:

```
public class Salut {
  public static void main (String args[]) {
    if (args.length == 0) {
      System.out.println("Numar insuficient de argumente!");
      System.exit(-1); //termina aplicatia
    }
    String nume = args[0]; //exista sigur
    String prenume;
    if (args.length >= 1)
      prenume = args[1];
    else
      prenume = ""; //valoare implicita
    System.out.println("Salut " + nume + " " + prenume);
  }
}
```

Spre deosebire de limbajul C, vectorul primit de metoda `main` nu conţine pe prima poziţie numele aplicaţiei, întrucât în Java numele aplicaţiei este chiar numele clasei principale, adică a clasei în care se găseşte metoda `main`.

Să consideră în continuare un exemplu simplu în care se doreşte afişarea pe ecran a argumentelor primite de la linia de comandă:

```
public class Afisare {
  public static void main (String[] args) {
    for (int i = 0; i < args.length; i++)
      System.out.println(args[i]);
```

```
  }
}
```

Un apel de genul `java Afisare Hello Java` va produce următorul rezultat (aplicaţia a primit 2 argumente):

```
Hello
Java
```

Apelul `java Afisare "Hello Java"` va produce însă alt rezultat (aplicaţia a primit un singur argument):

```
Hello Java
```

1.8.3 Argumente numerice

Argumentele de la linia de comandă sunt primite sub forma unui vector de şiruri (obiecte de tip `String`). In cazul în care unele dintre acestea reprezintă valori numerice ele vor trebui convertite din şiruri în numere. Acest lucru se realizează cu metode de tipul `parse`*TipNumeric* aflate în clasa corespunzatoare tipului în care vrem să facem conversia: `Integer, Float, Double`, etc.

Să considerăm, de exemplu, că aplicaţia `Power` ridică un numar real la o putere întreagă, argumentele fiind trimise de la linia de comandă sub forma:

```
java Power "1.5" "2" //ridica 1.5 la puterea 2
```

Conversia celor două argumente în numere se va face astfel:

```
public class Power {
  public static void main(String args[]) {
    double numar = Double.parseDouble(args[0]);
    int putere = Integer.parseInt(args[1]);
    System.out.println("Rezultat=" + Math.pow(numar, putere));
  }
}
```

Metodele de tipul `parse`*TipNumeric* pot produce excepţii (erori) de tipul `NumberFormatException` în cazul în care şirul primit ca parametru nu reprezintă un numar de tipul respectiv. Tratarea acestor excepţii este prezentată în capitolul "Excepţii".

Capitolul 2

Obiecte şi clase

2.1 Ciclul de viaţă al unui obiect

2.1.1 Crearea obiectelor

In Java, ca în orice limbaj de programare orientat-obiect, crearea obiectelor se realizează prin *instanţierea* unei clase şi implică următoarele lucruri:

- **Declararea**
  Presupune specificarea tipului acelui obiect, cu alte cuvinte specificarea clasei acestuia (vom vedea că tipul unui obiect poate fi şi o interfaţă).

  ```
  NumeClasa numeObiect;
  ```

- **Instanţierea**
  Se realizează prin intermediul operatorului **new** şi are ca efect crearea efectivă a obiectului cu alocarea spaţiului de memorie corespunzător.

  ```
  numeObiect = new NumeClasa();
  ```

- **Iniţializarea**
  Se realizează prin intermediul constructorilor clasei respective. Iniţializarea este de fapt parte integrantă a procesului de instanţiere, în sensul că imediat după alocarea memoriei ca efect al operatorului `new` este apelat constructorul specificat. Parantezele rotunde de după numele clasei indică faptul că acolo este de fapt un apel la unul din constructorii clasei şi nu simpla specificare a numelui clasei.
  Mai general, instanţierea şi iniţializarea apar sub forma:

```
numeObiect = new NumeClasa([argumente constructor]);
```

Să considerăm următorul exemplu, în care declarăm şi instanţiem două obiecte din clasa `Rectangle`, clasă ce descrie suprafeţe grafice rectangulare, definite de coordonatele colţului stânga sus (originea) şi lăţimea, respectiv înălţimea.

```
Rectangle r1, r2;
r1 = new Rectangle();
r2 = new Rectangle(0, 0, 100, 200);
```

In primul caz `Rectangle()` este un apel către constructorul clasei `Rectangle` care este responsabil cu iniţializarea obiectului cu valorile implicite. După cum observăm în al doilea caz, iniţializarea se poate face şi cu anumiţi parametri, cu condiţia să existe un constructor al clasei respective care să accepte parametrii respectivi.

Fiecare clasă are un set de constructori care se ocupă cu iniţializare obiectelor nou create. De exemplu, clasa `Rectangle` are următorii constructori:

```
public Rectangle()
public Rectangle(int latime, int inaltime)
public Rectangle(int x, int y, int latime, int inaltime)
public Rectangle(Point origine)
public Rectangle(Point origine, int latime, int inaltime)
public Rectangle(Point origine, Dimension dimensiune)
```

Declararea, instanţierea şi iniţializarea obiectului pot apărea pe aceeaşi linie (cazul cel mai uzual):

```
Rectangle patrat = new Rectangle(0, 0, 100, 100);
```

Obiecte anonime

Este posibilă şi crearea unor obiecte anonime care servesc doar pentru iniţializarea altor obiecte, caz în care etapa de declarare a referinţei obiectului nu mai este prezentă:

```
Rectangle patrat = new Rectangle(new Point(0,0),
                                 new Dimension(100, 100));
```

Spaţiul de memorie nu este pre-alocat

Declararea unui obiect nu implică sub nici o formă alocarea de spaţiu de memorie pentru acel obiect. Alocarea memoriei se face doar la apelul operatorului `new`.

```
Rectangle patrat;
patrat.x = 10;
//Eroare - lipseste instantierea
```

2.1.2 Folosirea obiectelor

Odată un obiect creat, el poate fi folosit în următoarele sensuri: aflarea unor informaţii despre obiect, schimbarea stării sale sau executarea unor acţiuni. Aceste lucruri se realizeaza prin aflarea sau schimbarea valorilor variabilelor sale, respectiv prin apelarea metodelor sale.

Referirea valorii unei variabile se face prin **obiect.variabila** De exemplu clasa `Rectangle` are variabilele publice `x, y, width, height, origin`. Aflarea valorilor acestor variabile sau schimbarea lor se face prin construcţii de genul:

```
Rectangle patrat = new Rectangle(0, 0, 100, 200);
System.out.println(patrat.width);  //afiseaza 100
patrat.x = 10;
patrat.y = 20;      //schimba originea
patrat.origin = new Point(10, 20); //schimba originea
```

Accesul la variabilele unui obiect se face în conformitate cu drepturile de acces pe care le oferă variabilele respective celorlalte clase. (vezi "Modificatori de acces pentru membrii unei clase")

Apelul unei metode se face prin **obiect.metoda([parametri])**.

```
Rectangle patrat = new Rectangle(0, 0, 100, 200);
patrat.setLocation(10, 20); //schimba originea
patrat.setSize(200, 300); //schimba dimensiunea
```

Se observă că valorile variabilelor pot fi modificate indirect prin intermediul metodelor sale. Programarea orientată obiect descurajează folosirea directă a variabilelor unui obiect deoarece acesta poate fi adus în stări inconsistente (ireale). In schimb, pentru fiecare variabilă care descrie starea

obiectului trebuie să existe metode care să permită schimbarea/aflarea valorilor variabilelor sale. Acestea se numesc *metode de accesare*, sau metode *setter - getter* şi au numele de forma `set`*Variabila*, respectiv `get`*Variabila*.

```
patrat.width = -100;         //stare inconsistenta
patrat.setSize(-100, -200);  //metoda setter
//metoda setSize poate sa testeze daca noile valori sunt
//corecte si sa valideze sau nu schimbarea lor
```

2.1.3 Distrugerea obiectelor

Multe limbaje de programare impun ca programatorul să ţină evidenţa obiectelor create şi să le distrugă în mod explicit atunci când nu mai este nevoie de ele, cu alte cuvinte să administreze singur memoria ocupată de obiectele sale. Practica a demonstrat că această tehnică este una din principalele furnizoare de erori ce duc la funcţionarea defectuoasă a programelor.

In Java programatorul nu mai are responsabilitatea distrugerii obiectelor sale întrucât, în momentul rulării unui program, simultan cu interpretorul Java, rulează şi un proces care se ocupă cu distrugerea obiectelor care nu mai sunt folosite. Acest proces pus la dispoziţie de platforma Java de lucru se numeşte **garbage collector** (colector de gunoi), prescurtat **gc**.

Un obiect este eliminat din memorie de procesul de colectare atunci când nu mai există nici o referinţă la acesta. Referinţele (care sunt de fapt variabile) sunt distruse două moduri:

- *natural*, atunci când variabila respectivă iese din domeniul său de vizibilitate, de exemplu la terminarea metodei în care ea a fost declarată;
- *explicit*, dacă atribuim variabilei respective valoare `null`.

Cum funcţionează colectorul de gunoaie ?

Colectorul de gunoaie este un proces de prioritate scazută care se execută periodic, scanează dinamic memoria ocupată de programul Java aflat în execuţie şi marchează acele obiecte care au referinţe directe sau indirecte. După ce toate obiectele au fost parcurse, cele care au rămas nemarcate sunt eliminate automat din memorie.

Apelul metodei `gc` din clasa `System` sugerează maşinii virtuale Java să "depună eforturi" în recuperarea memoriei ocupate de obiecte care nu mai sunt folosite, fără a forţa însă pornirea procesului.

Finalizare

Inainte ca un obiect să fie eliminat din memorie, procesul gc dă acelui obiect posibilitatea "să cureţe după el", apelând metoda de finalizare a obiectului respectiv. Uzual, în timpul finalizării un obiect îşi inchide fisierele şi socket-urile folosite, distruge referinţele către alte obiecte (pentru a uşsura sarcina colectorului de gunoaie), etc.

Codul pentru finalizarea unui obiect trebuie scris într-o metodă specială numita `finalize` a clasei ce descrie obiectul respectiv. (vezi "Clasa Object")

Atenţie

Nu confundati metoda finalize din Java cu destructorii din C++. Metoda finalize nu are rolul de a distruge obiectul ci este apelată automat înainte de eliminarea obiectului respectiv din memorie.

2.2 Crearea claselor

2.2.1 Declararea claselor

Clasele reprezintă o modalitate de a introduce noi tipuri de date într-o aplicaţie Java, cealaltă modalitate fiind prin intermediul interfeţelor. Declararea unei clase respectă următorul format general:

```
[public][abstract][final]class NumeClasa
  [extends NumeSuperclasa]
  [implements Interfata1 [, Interfata2 ... ]]
  {
    // Corpul clasei
  }
```

Aşadar, prima parte a declaraţiei o ocupă modificatorii clasei. Aceştia sunt:

- **public**
  Implicit, o clasă poate fi folosită doar de clasele aflate în acelaşi pachet(librărie) cu clasa respectivă (dacă nu se specifică un anume pachet, toate clasele din directorul curent sunt considerate a fi în acelaşi pachet). O clasă declarată cu `public` poate fi folosită din orice altă clasă, indiferent de pachetul în care se găseşte.

- **abstract**
  Declară o clasă abstractă (şablon). O clasă abstractă nu poate fi instanţiată, fiind folosită doar pentru a crea un model comun pentru o serie de subclase. (vezi "Clase şi metode abstracte")

- **final**
  Declară că respectiva clasă nu poate avea subclase. Declarare claselor finale are două scopuri:

  - *securitate*: unele metode pot aştepta ca parametru un obiect al unei anumite clase şi nu al unei subclase, dar tipul exact al unui obiect nu poate fi aflat cu exactitate decat în momentul executiei; în felul acesta nu s-ar mai putea realiza obiectivul limbajului Java ca un program care a trecut compilarea să nu mai fie susceptibil de nici o eroare.
  - *programare în spririt orientat-obiect*: O clasa "perfectă" nu trebuie să mai aibă subclase.

După numele clasei putem specifica, dacă este cazul, faptul că respectiva clasă este subclasă a unei alte clase cu numele *NumeSuperclasa* sau/şi că implementează una sau mai multe interfeţe, ale căror nume trebuie separate prin virgulă.

2.2.2 Extinderea claselor

Spre deosebire de alte limbaje de programare orientate-obiect, Java permite doar **moştenirea simplă**, ceea ce înseamnă că o clasă poate avea un singur părinte (superclasă). Evident, o clasă poate avea oricâti moştenitori (subclase), de unde rezultă că mulţimea tuturor claselor definite în Java poate fi vazută ca un arbore, rădăcina acestuia fiind clasa `Object`. Aşadar, `Object` este singura clasă care nu are părinte, fiind foarte importantă în modul de lucru cu obiecte si structuri de date în Java.

Extinderea unei clase se realizează folosind cuvântul cheie **extends**:

```
class B extends A {...}
// A este superclasa clasei B
// B este o subclasa a clasei A
```

O subclasă moşteneşte de la părintele său toate variabilele şi metodele care nu sunt private.

2.2.3 Corpul unei clase

Corpul unei clase urmează imediat după declararea clasei şi este cuprins între acolade. Conţinutul acestuia este format din:

- Declararea şi, eventual, iniţializarea variabilelor de instanţă şi de clasă (cunoscute împreună ca *variabile membre*).
- Declararea şi implementarea constructorilor.
- Declararea şi implementarea metodelor de instanţa şi de clasă (cunoscute împreună ca *metode membre*).
- Declararea unor clase imbricate (interne).

Spre deosebire de C++, nu este permisă doar declararea metodei în corpul clasei, urmând ca implementare să fie facută în afara ei. Implementarea metodelor unei clase trebuie să se facă obligatoriu în corpul clasei.

```
// C++
class A {
  void metoda1();
  int  metoda2()  {
    // Implementare
  }
}
A::metoda1() {
  // Implementare
}
```

```
// Java
class A {
  void metoda1(){
    // Implementare
  }
  void metoda2(){
    // Implementare
  }
}
```

Variabilele unei clase pot avea acelaşi nume cu metodele clasei, care poate fi chiar numele clasei, fără a exista posibilitatea apariţiei vreunei ambiguităţi din punctul de vedere al compilatorului. Acest lucru este însă total nerecomandat dacă ne gândim din perspectiva lizibilităţii (claritătii) codului, dovedind un stil ineficient de progamare.

```
class A {
  int A;
  void A() {};
  // Corect pentru compilator
  // Nerecomandat ca stil de programare
}
```

Atenţie
Variabilele şi metodele nu pot avea ca nume un cuvânt cheie Java.

2.2.4 Constructorii unei clase

Constructorii unei clase sunt metode speciale care au acelaşi nume cu cel al clasei, nu returnează nici o valoare şi sunt folosiţi pentru iniţializarea obiectelor acelei clase în momentul instanţierii lor.

```
class NumeClasa {
  [modificatori] NumeClasa([argumente]) {
    // Constructor
```

```
  }
}
```

O clasă poate avea unul sau mai mulţi constructori care trebuie însă să difere prin lista de argumente primite. In felul acesta sunt permise diverse tipuri de iniţializări ale obiectelor la crearea lor, în funcţie de numărul parametrilor cu care este apelat constructorul.

Să considerăm ca exemplu declararea unei clase care descrie noţiunea de dreptunghi şi trei posibili constructori pentru aceasta clasă.

```
class Dreptunghi {
  double x, y, w, h;
  Dreptunghi(double x1, double y1, double w1, double h1) {
    // Cel mai general constructor
    x=x1; y=y1; w=w1; h=h1;
    System.out.println("Instantiere dreptunghi");
  }
  Dreptunghi(double w1, double h1) {
    // Constructor cu doua argumente
    x=0; y=0; w=w1; h=h1;
    System.out.println("Instantiere dreptunghi");
  }

  Dreptunghi() {
    // Constructor fara argumente
    x=0; y=0; w=0; h=0;
    System.out.println("Instantiere dreptunghi");
  }
}
```

Constructorii sunt apelaţi automat la instanţierea unui obiect. In cazul în care dorim să apelăm explicit constructorul unei clase folosim expresia

```
this( argumente ),
```

care apelează constructorul corespunzător (ca argumente) al clasei respective. Această metodă este folosită atunci când sunt implementaţi mai mulţi constructori pentru o clasă, pentru a nu repeta secvenţele de cod scrise deja la constructorii cu mai multe argumente (mai generali). Mai eficient, fără

a repeta aceleaşi secvenţe de cod în toţi constructorii (cum ar fi afişarea mesajului "Instantiere dreptunghi"), clasa de mai sus poate fi rescrisă astfel:

```
class Dreptunghi {
  double x, y, w, h;
  Dreptunghi(double x1, double y1, double w1, double h1) {
    // Implementam doar constructorul cel mai general
    x=x1; y=y1; w=w1; h=h1;
    System.out.println("Instantiere dreptunghi");
  }
  Dreptunghi(double w1, double h1) {
    this(0, 0, w1, h1);
    // Apelam constructorul cu 4 argumente
  }

  Dreptunghi() {
    this(0, 0);
    // Apelam constructorul cu 2 argumente
  }
}
```

Dintr-o subclasă putem apela explicit constructorii superclasei cu expresia

`super( argumente )`.

Să presupunem că dorim să creăm clasa `Patrat`, derivată din clasa `Dreptunghi`:

```
class Patrat extends Dreptunghi {
  Patrat(double x, double y, double d) {
    super(x, y, d, d);
    // Apelam constructorul superclasei
  }
}
```

Atenţie

Apelul explcit al unui constructor nu poate apărea decât într-un alt constructor si trebuie să fie prima instrucţiune din constructorul respectiv.

Constructorul implicit

Constructorii sunt apelaţi automat la instanţierea unui obiect. In cazul în care scriem o clasă care nu are declarat nici un constructor, sistemul îi creează automat un constructor implicit, care nu primeşte nici un argument şi care nu face nimic. Deci prezenţa constructorilor în corpul unei clase nu este obligatorie. Dacă însă scriem un constructor pentru o clasă, care are mai mult de un argument, atunci constructorul implicit (fără nici un argument) nu va mai fi furnizat implicit de către sistem. Să considerăm, ca exemplu, următoarele declaraţii de clase:

```
class Dreptunghi {
  double x, y, w, h;
  // Nici un constructor
}
class Cerc {
  double x, y, r;
  // Constructor cu 3 argumente
  Cerc(double x, double y, double r) { ... };
}
```

Să considerăm acum două instanţieri ale claselor de mai sus:

```
Dreptunghi d = new Dreptunghi();
// Corect (a fost generat constructorul implicit)

Cerc c;
c = new Cerc();
// Eroare la compilare !

c = new Cerc(0, 0, 100);
// Varianta corecta
```

In cazul moştenirii unei clase, instanţierea unui obiect din clasa extinsă implică instanţierea unui obiect din clasa părinte. Din acest motiv, fiecare constructor al clasei fiu va trebui să aibă un constructor cu aceeaşi signatură în părinte sau să apeleze explicit un constructor al clasei extinse folosind expresia `super([argumente])`, în caz contrar fiind semnalată o eroare la compilare.

```
class A {
  int x=1;
  A(int x) { this.x = x;}
}
class B extends A {
  // Corect
  B() {super(2);}
  B(int x) {super.x = x;}
}

class C extends A {
  // Eroare la compilare !
  C() {super.x = 2;}
  C(int x) {super.x = x;}
}
```

Constructorii unei clase pot avea următorii modificatori de acces: `public, protected, private` şi cel implicit.

- **public**
  In orice altă clasă se pot crea instanţe ale clasei respective.
- **protected**
  Doar în subclase pot fi create obiecte de tipul clasei respective.
- **private**
  In nici o altă clasă nu se pot instanţia obiecte ale acestei clase. O astfel de clasă poate conţine metode publice (numite "factory methods") care să fie responsabile cu crearea obiectelor, controlând în felul acesta diverse aspecte legate de instanţierea clasei respective.
- **implicit**
  Doar în clasele din acelaşi pachet se pot crea instanţe ale clasei respective.

2.2.5 Declararea variabilelor

Variabilele membre ale unei clase se declară de obicei înaintea metodelor, deşi acest lucru nu este impus de către compilator.

```
  class NumeClasa {
  // Declararea variabilelor
  // Declararea metodelor
}
```

Variabilele membre ale unei clase se declară în corpul clasei şi nu în corpul unei metode, fiind vizibile în toate metodele respectivei clase. Variabilele declarate în cadrul unei metode sunt locale metodei respective.

Declararea unei variabile presupune specificarea următoarelor lucruri:

- numele variabilei
- tipul de date al acesteia
- nivelul de acces la acea variabila din alte clase
- dacă este constantă sau nu
- dacă este variabilă de instanţă sau de clasă
- alţi modificatori

Generic, o variabilă se declară astfel:

```
  [modificatori] Tip numeVariabila [ = valoareInitiala ];
```

unde un modificator poate fi :

- un modificator de acces : `public, protected, private` (vezi "Modificatori de acces pentru membrii unei clase")
- unul din cuvintele rezervate: `static, final, transient, volatile`

Exemple de declaraţii de variabile membre:

```
class Exemplu {
  double x;
  protected static int n;
  public String s = "abcd";
  private Point p = new Point(10, 10);
  final static long MAX = 100000L;
}
```

Să analizăm modificatorii care pot fi specificaţi pentru o variabilă, alţii decât cei de acces care sunt tratati într-o secţiune separata: "Specificatori de acces pentru membrii unei clase".

- **static**
  Prezenţa lui declară că o variabilă este variabilă de clasă şi nu de instanţă. (vezi "Membri de instanţa şi membri de clasă")

```
int variabilaInstanta ;
static int variabilaClasa;
```

- **final**
  Indică faptul că valoarea variabilei nu mai poate fi schimbată, cu alte cuvinte este folosit pentru declararea constantelor.

```
final double PI = 3.14 ;
...
PI = 3.141; // Eroare la compilare !
```

  Prin convenţie, numele variabilelor finale se scriu cu litere mari. Folosirea lui `final` aduce o flexibilitate sporită în lucrul cu constante, în sensul că valoarea unei variabile nu trebuie specificată neapărat la declararea ei (ca în exemplul de mai sus), ci poate fi specificată şi ulterior într-un constructor, după care ea nu va mai putea fi modificată.

```
class Test {
  final int MAX;
  Test() {
    MAX = 100; // Corect
    MAX = 200; // Eroare la compilare !
  }
}
```

- **transient**
  Este folosit la serializarea obiectelor, pentru a specifica ce variabile membre ale unui obiect nu participă la serializare. (vezi "Serializarea obiectelor")

- **volatile**
  Este folosit pentru a semnala compilatorului să nu execute anumite optimizări asupra membrilor unei clase. Este o facilitate avansată a limbajului Java.

2.2.6 this şi super

Sunt variabile predefinite care fac referinţa, în cadrul unui obiect, la obiectul propriu-zis (`this`), respectiv la instanţa părintelui (`super`). Sunt folosite în general pentru a rezolva conflicte de nume prin referirea explicită a unei variabile sau metode membre. După cum am văzut, utilizate sub formă de metode au rolul de a apela constructorii corespunzători ca argumente ai clasei curente, respectiv ai superclasei

```
class A {
  int x;
  A() {
    this(0);
  }
  A(int x) {
    this.x = x;
  }
  void metoda() {
    x ++;
  }
}

class B extends A {
  B() {
    this(0);
  }
  B(int x) {
    super(x);
    System.out.println(x);
  }

  void metoda() {
    super.metoda();
```

```
    System.out.println(x);
  }
}
```

2.3 Implementarea metodelor

2.3.1 Declararea metodelor

Metodele sunt responsabile cu descrierea comportamentului unui obiect. Întrucât Java este un limbaj de programare complet orientat-obiect, metodele se pot găsi doar în cadrul claselor. Generic, o metodă se declară astfel:

```
[modificatori] TipReturnat numeMetoda ( [argumente] )
  [throws TipExceptie1, TipExceptie2, ...]
  {
    // Corpul metodei
  }
```

unde un modificator poate fi :

- un specificator de acces : `public, protected, private` (vezi "Specificatori de acces pentru membrii unei clase")
- unul din cuvintele rezervate: `static, abstract, final, native, synchronized`

Să analizăm modificatorii care pot fi specificaţi pentru o metodă, alţii decât cei de acces care sunt trataţi într-o secţiune separată.

- **static**
  Prezenţa lui declară că o metodă este de clasă şi nu de instanţă. (vezi "Membri de instanţa şi membri de clasă")

  ```
  void metodaInstanta();
  static void metodaClasa();
  ```

- **abstract**
  Permite declararea metodelor abstracte. O metodă abstractă este o metodă care nu are implementare şi trebuie obligatoriu să facă parte dintr-o clasă abstractă. (vezi "Clase şi metode abstracte")

- **final**
  Specifică faptul că acea metoda nu mai poate fi supradefinită în subclasele clasei în care ea este definită ca fiind finală. Acest lucru este util dacă respectiva metodă are o implementare care nu trebuie schimbată sub nici o formă în subclasele ei, fiind critică pentru consistenţa stării unui obiect. De exemplu, studenţilor unei universităţi trebuie să li se calculeze media finala, în funcţie de notele obţinute la examene, în aceeaşi manieră, indiferent de facultatea la care sunt.

```
class Student {
  ...
  final float calcMedie(float note[], float ponderi[]) {
  ...
  }
  ...
}
class StudentInformatica extends Student {
  float calcMedie(float note[], float ponderi[]) {
    return 10.00;
  }
}// Eroare la compilare !
```

- **native**
  In cazul în care avem o librărie importantă de funcţii scrise în alt limbaj de programare, cum ar fi C, C++ şi limbajul de asamblare, acestea pot fi refolosite din programele Java. Tehnologia care permite acest lucru se numeşte *JNI (Java Native Interface)* şi permite asocierea dintre metode Java declarate cu `native` şi metode native scrise în limbajele de programare menţionate.

- **synchronized**
  Este folosit în cazul în care se lucrează cu mai multe fire de execuţie iar metoda respectivă gestionează resurse comune. Are ca efect construirea unui monitor care nu permite executarea metodei, la un moment dat, decât unui singur fir de execuţie. (vezi "Fire de executie")

2.3.2 Tipul returnat de o metodă

Metodele pot sau nu să returneze o valoare la terminarea lor. Tipul returnat poate fi atât un tip primitiv de date sau o referinţă la un obiect al unei clase. In cazul în care o metodă nu returnează nimic atunci trebuie obligatoriu specificat cuvântul cheie `void` ca tip returnat:

```
public  void afisareRezultat() {
  System.out.println("rezultat");
}
private void deseneaza(Shape s) {
  ...
  return;
}
```

Dacă o metodă trebuie să returneze o valoare acest lucru se realizează prin intermediul instrucţiunii `return`, care trebuie să apară în toate situaţiile de terminare a funcţiei.

```
double radical(double x) {
  if (x >= 0)
    return Math.sqrt(x);
  else {
    System.out.println("Argument negativ !");
    // Eroare la compilare
    // Lipseste return pe aceasta ramura
  }
}
```

In cazul în care în declaraţia funcţiei tipul returnat este un tip primitiv de date, valoarea returnată la terminarea funcţiei trebuie să aibă obligatoriu acel tip sau un subtip al său, altfel va fi furnizată o eroare la compilare. In general, orice atribuire care implică pierderi de date este tratată de compilator ca eroare.

```
int metoda() {
  return 1.2; // Eroare
}

int metoda() {
```

```
  return (int)1.2; // Corect
}

double metoda() {
  return (float)1; // Corect
}
```

Dacă valoarea returnată este o referinţă la un obiect al unei clase, atunci clasa obiectului returnat trebuie să coincidă sau să fie o subclasă a clasei specificate la declararea metodei. De exemplu, fie clasa `Poligon` şi subclasa acesteia `Patrat`.

```
Poligon metoda1( ) {
  Poligon p = new Poligon();
  Patrat t = new Patrat();
  if (...)
    return p; // Corect
  else
    return t; // Corect
}
Patrat metoda2( ) {
  Poligon p = new Poligon();
  Patrat t = new Patrat();
  if (...)
    return p; // Eroare
  else
    return t; // Corect
}
```

2.3.3 Trimiterea parametrilor către o metodă

Signatura unei metode este dată de numarul şi tipul argumentelor primite de acea metodă. Tipul de date al unui argument poate fi orice tip valid al limbajului Java, atât tip primitiv cât şi tip referinţă.

```
TipReturnat metoda([Tip1 arg1, Tip2 arg2, ...])
```

Exemplu:

```
  void adaugarePersoana(String nume, int varsta, float salariu)
  // String este tip referinta
  // int si float sunt tipuri primitive
```

Spre deosebire de alte limbaje, în Java nu pot fi trimise ca parametri ai unei metode referinţe la alte metode (funcţii), însă pot fi trimise referinţe la obiecte care să conţină implementarea acelor metode, pentru a fi apelate.
Pâna la apariţia versiunii 1.5, în Java o metodă nu putea primi un număr variabil de argumente, ceea ce înseamna că apelul unei metode trebuia să se facă cu specificarea exactă a numarului şi tipurilor argumentelor. Vom analiza într-o secţiune separată modalitate de specificare a unui număr variabil de argumente pentru o metodă.
Numele argumentelor primite trebuie să difere între ele şi nu trebuie să coincidă cu numele nici uneia din variabilele locale ale metodei. Pot însă să coincidă cu numele variabilelor membre ale clasei, caz în care diferenţierea dintre ele se va face prin intermediul variabile `this`.

```
class Cerc {
  int x, y, raza;
  public Cerc(int x, int y, int raza) {
    this.x = x;
    this.y = y;
    this.raza = raza;
  }
}
```

In Java argumentele sunt trimise **doar prin valoare** (pass-by-value). Acest lucru înseamnă că metoda recepţionează doar valorile variabilelor primite ca parametri.
Când argumentul are tip primitiv de date, metoda nu-i poate schimba valoarea decât local (în cadrul metodei); la revenirea din metodă variabila are aceeaşi valoare ca înaintea apelului, modificările făcute în cadrul metodei fiind pierdute.
Când argumentul este de tip referinţă, metoda nu poate schimba valoarea referinţei obiectului, însă poate apela metodele acelui obiect şi poate modifica orice variabilă membră accesibilă.

Aşadar, dacă dorim ca o metodă să schimbe starea (valoarea) unui argument primit, atunci el trebuie să fie neaparat de tip referinţă.

De exemplu, să considerăm clasa `Cerc` descrisă anterior în care dorim să implementăm o metodă care să returneze parametrii cercului.

```
// Varianta incorecta:
class Cerc {
  private int x, y, raza;
  public void aflaParametri(int valx, int valy, int valr) {
    // Metoda nu are efectul dorit!
    valx = x;
    valy = y;
    valr = raza;
  }
}
```

Această metodă nu va realiza lucrul propus întrucât ea primeşte doar valorile variabilelor `valx, valy şi valr` şi nu referinţe la ele (adresele lor de memorie), astfel încât să le poată modifica valorile. In concluzie, metoda nu realizează nimic pentru că nu poate schimba valorile variabilelor primite ca argumente.

Pentru a rezolva lucrul propus trebuie să definim o clasă suplimentară care să descrie parametrii pe care dorim să-i aflăm:

```
// Varianta corecta
class Param {
  public int x, y, raza;
}

class Cerc {
  private int x, y, raza;
  public void aflaParametri(Param param) {
    param.x = x;
    param.y = y;
    param.raza = raza;
  }
}
```

Argumentul `param` are tip referinţă şi, deşi nu îi schimbăm valoarea (valoarea sa este adresa de memorie la care se găseşte şi nu poate fi schimbată),

putem schimba starea obiectului, adică informaţia propriu-zisă conţinută de acesta.

Varianta de mai sus a fost dată pentru a clarifica modul de trimitere a argumentelor unei metode. Pentru a afla însă valorile variabilelor care descriu starea unui obiect se folosesc metode de tip *getter* însoţite de metode *setter* care să permită schimbarea stării obiectului:

```
class Cerc {
  private int x, y, raza;
  public int getX() {
    return x;
  }
  public void setX(int x) {
    this.x = x;
  }
  ...
}
```

2.3.4 Metode cu număr variabil de argumente

Incepând cu versiunea 1.5 a limbajului Java, există posibilitate de a declara metode care să primească un număr variabil de argumente. Noutatea constă în folosirea simbolului **...**, sintaxa unei astfel de metode fiind:

```
[modificatori] TipReturnat metoda(TipArgumente ...  args)
```

args reprezintă un vector având tipul specificat şi instanţiat cu un număr variabil de argumente, în funcţie de apelul metodei. Tipul argumentelor poate fi referinţă sau primitiv. Metoda de mai jos afişează argumentele primite, care pot fi de orice tip:

```
void metoda(Object ... args) {
  for(int i=0; i<args.length; i++)
    System.out.println(args[i]);
}
...
metoda("Hello");
metoda("Hello", "Java", 1.5);
```

2.3.5 Supraîncărcarea şi supradefinirea metodelor

Supraîncărcarea şi supradefinirea metodelor sunt două concepte extrem de utile ale programării orientate obiect, cunoscute şi sub denumirea de *polimorfism*, şi se referă la:

- *supraîncarcarea (overloading)* : în cadrul unei clase pot exista metode cu acelaşi nume cu condiţia ca signaturile lor să fie diferite (lista de argumente primite să difere fie prin numărul argumentelor, fie prin tipul lor) astfel încât la apelul funcţiei cu acel nume să se poată stabili în mod unic care dintre ele se execută.
- *supradefinirea (overriding)*: o subclasă poate rescrie o metodă a clasei părinte prin implementarea unei metode cu acelaşi nume şi aceeaşi signatură ca ale superclasei.

```
class A {
  void metoda() {
    System.out.println("A: metoda fara parametru");
  }
  // Supraincarcare
  void metoda(int arg) {
    System.out.println("A: metoda cu un parametru");
  }
}

class B extends A {
  // Supradefinire
  void metoda() {
    System.out.println("B: metoda fara parametru");
  }
}
```

O metodă supradefinită poate să:

- **ignore** complet codul metodei corespunzătoare din superclasă (cazul de mai sus):

```
B b = new B();
b.metoda();
// Afiseaza "B: metoda fara parametru"
```

- **extindă** codul metodei părinte, executând înainte de codul propriu şi funcţia părintelui:

```
class B extends A {
  // Supradefinire prin extensie
  void metoda() {
    super.metoda();
    System.out.println("B: metoda fara parametru");
  }
}
  . . .
  B b = new B();
  b.metoda();
  /* Afiseaza ambele mesaje:
   "A: metoda fara parametru"
   "B: metoda fara parametru" */
```

O metodă nu poate supradefini o metodă declarată finală în clasa părinte.

Orice clasă care nu este abstractă trebuie obligatoriu să supradefinească metodele abstracte ale superclasei (dacă este cazul). In cazul în care o clasă nu supradefineşte toate metodele abstracte ale părintelui, ea însăşi este abstractă şi va trebui declarată ca atare.

In Java nu este posibilă supraîncărcarea operatorilor.

2.4 Modificatori de acces

Modificatorii de acces sunt cuvinte rezervate ce controlează accesul celorlate clase la membrii unei clase. Specificatorii de acces pentru variabilele şi metodele unei clase sunt: **public, protected, private** şi cel implicit (la nivel de pachet), iar nivelul lor de acces este dat în tabelul de mai jos:

Specificator	Clasa	Sublasa	Pachet	Oriunde
`private`	X			
`protected`	X	X*	X	
`public`	X	X	X	X
implicit	X		X	

Aşadar, dacă nu este specificat nici un modificator de acces, implicit nivelul de acces este la nivelul pachetului. In cazul în care declarăm un membru "protected" atunci accesul la acel membru este permis din subclasele clasei în care a fost declarat dar depinde şi de pachetul în care se gaseşte subclasa: dacă sunt în acelaşi pachet accesul este permis, dacă nu sunt în acelaşi pachet accesul nu este permis decât pentru obiecte de tipul subclasei.

Exemple de declaraţii:

```
private int secretPersonal;
protected String secretDeFamilie;
public Vector pentruToti;
long doarIntrePrieteni;
private void metodaInterna();
public String informatii();
```

2.5 Membri de instanţă şi membri de clasă

O clasă Java poate conţine două tipuri de variabile şi metode :

- *de instanţă*: declarate **fără** modificatorul `static`, specifice fiecărei instanţe create dintr-o clasă şi
- *de clasă*: declarate **cu** modificatorul `static`, specifice clasei.

2.5.1 Variabile de instanţă şi de clasă

Când declarăm o variabilă membră fără modificatorul `static`, cum ar fi `x` în exemplul de mai jos:

```
class Exemplu {
  int x ; //variabila de instanta
}
```

se declară de fapt o variabilă de instanţă, ceea ce înseamnă că la fiecare creare a unui obiect al clasei `Exemplu` sistemul alocă o zonă de memorie separată pentru memorarea valorii lui x.

```
Exemplu o1 = new Exemplu();
o1.x = 100;
```

```
  Exemplu o2 = new Exemplu();
  o2.x = 200;
  System.out.println(o1.x); // Afiseaza 100
  System.out.println(o2.x); // Afiseaza 200
```

Aşadar, fiecare obiect nou creat va putea memora valori diferite pentru variabilele sale de instanţă.

Pentru variabilele de clasă (statice) sistemul alocă o singură zonă de memorie la care au acces toate instanţele clasei respective, ceea ce înseamnă că dacă un obiect modifică valoarea unei variabile statice ea se va modifica şi pentru toate celelalte obiecte. Deoarece nu depind de o anumită instanţă a unei clase, variabilele statice pot fi referite şi sub forma:

```
NumeClasa.numeVariabilaStatica
```

```
class Exemplu {
  int x ;         // Variabila de instanta
  static long n;  // Variabila de clasa
}
  . . .
  Exemplu o1 = new Exemplu();
  Exemplu o2 = new Exemplu();
  o1.n = 100;
  System.out.println(o2.n);       // Afiseaza 100
  o2.n = 200;
  System.out.println(o1.n);       // Afiseaza 200
  System.out.println(Exemplu.n);  // Afiseaza 200
  // o1.n, o2.n si Exemplu.n sunt referinte la aceeasi valoare
```

Iniţializarea variabilelor de clasă se face o singură dată, la încărcarea în memorie a clasei respective, şi este realizată prin atribuiri obişnuite:

```
class Exemplu {
  static final double PI = 3.14;
  static long nrInstante = 0;
  static Point p = new Point(0,0);
}
```

2.5.2 Metode de instanţă şi de clasă

Similar ca la variabile, metodele declarate fără modificatorul `static` sunt metode de instanţă iar cele declarate cu `static` sunt metode de clasă (statice). Diferenţa între cele două tipuri de metode este următoarea:

- metodele de instanţă operează atât pe variabilele de instanţă cât şi pe cele statice ale clasei;
- metodele de clasă operează doar pe variabilele statice ale clasei.

```
class Exemplu {
  int x ;        // Variabila de instanta
  static long n; // Variabila de clasa
  void metodaDeInstanta() {
    n ++;  // Corect
    x --;  // Corect
  }
  static void metodaStatica() {
    n ++;  // Corect
    x --;  // Eroare la compilare !
  }
}
```

Intocmai ca şi la variabilele statice, întrucât metodele de clasă nu depind de starea obiectelor clasei respective, apelul lor se poate face şi sub forma:

```
NumeClasa.numeMetodaStatica
```

```
Exemplu.metodaStatica();     // Corect, echivalent cu
Exemplu obj = new Exemplu();
obj.metodaStatica();         // Corect, de asemenea
```

Metodele de instanţă nu pot fi apelate decât pentru un obiect al clasei respective:

```
Exemplu.metodaDeInstanta();  // Eroare la compilare !
Exemplu obj = new Exemplu();
obj.metodaDeInstanta();      // Corect
```

2.5.3 Utilitatea membrilor de clasă

Membrii de clasă sunt folosiţi pentru a pune la dispoziţie valori şi metode independente de starea obiectelor dintr-o anumita clasă.

Declararea eficientă a constantelor

Să considerăm situaţia când dorim să declarăm o constantă.

```
class Exemplu {
  final double PI = 3.14;
  // Variabila finala de instanta
}
```

La fiecare instanţiere a clasei `Exemplu` va fi rezervată o zonă de memorie pentru variabilele finale ale obiectului respectiv, ceea ce este o risipă întrucât aceste constante au aceleaşi valori pentru toate instanţele clasei. Declararea corectă a constantelor trebuie aşadar facută cu modificatorii `static` şi `final`, pentru a le rezerva o singură zonă de memorie, comună tuturor obiectelor:

```
class Exemplu {
  static final double PI = 3.14;
  // Variabila finala de clasa
}
```

Numărarea obiectelor unei clase

Numărarea obiectelor unei clase poate fi făcută extrem de simplu folosind o variabilă statică şi este utilă în situaţiile când trebuie să controlăm diverşi parametri legaţi de crearea obiectelor unei clase.

```
class Exemplu {
  static long nrInstante = 0;
  Exemplu() {
    // Constructorul este apelat la fiecare instantiere
    nrInstante ++;
  }
}
```

Implementarea funcţiilor globale

Spre deosebire de limbajele de programare procedurale, în Java nu putem avea funcţii globale definite ca atare, întrucât "orice este un obiect". Din acest motiv chiar şi metodele care au o funcţionalitate globală trebuie implementate în cadrul unor clase. Acest lucru se va face prin intermediul metodelor de clasă (globale), deoarece acestea nu depind de starea particulară a obiectelor din clasa respectivă. De exemplu, să considerăm funcţia `sqrt` care extrage radicalul unui număr şi care se găseşte în clasa `Math`. Dacă nu ar fi fost funcţie de clasă, apelul ei ar fi trebuit făcut astfel (incorect, de altfel):

```
// Incorect !
Math obj = new Math();
double rad = obj.sqrt(121);
```

ceea ce ar fi fost extrem de neplăcut... Fiind însă metodă statică ea poate fi apelată prin: `Math.sqrt(121)` .
Aşadar, funcţiile globale necesare unei aplicaţii vor fi grupate corespunzător în diverse clase şi implementate ca metode statice.

2.5.4 Blocuri statice de iniţializare

Variabilele statice ale unei clase sunt iniţializate la un moment care precede prima utilizare activă a clasei respective. Momentul efectiv depinde de implementarea maşinii virtuale Java şi poartă numele de *iniţializarea clasei*. Pe lângă setarea valorilor variabilelor statice, în această etapă sunt executate şi blocurile statice de iniţializare ale clasei. Acestea sunt secvenţe de cod de forma:

```
static {
  // Bloc static de initializare;
  ...
}
```

care se comportă ca o metodă statică apelată automat de către maşina virtuală. Variabilele referite într-un bloc static de iniţializare trebuie să fie obligatoriu de clasă sau locale blocului:

```
public class Test {
  // Declaratii de variabile statice
```

```
  static int x = 0, y, z;

  // Bloc static de initializare
  static {
    System.out.println("Initializam...");
    int t=1;
    y = 2;
    z = x + y + t;
  }

  Test() {
    /* La executia constructorului
       variabilele de clasa sunt deja initializate si
       toate blocurile statice de initializare au
       fost obligatoriu executate in prealabil.
    */
    ...
  }
}
```

2.6 Clase imbricate

2.6.1 Definirea claselor imbricate

O *clasă imbricată* este, prin definiţie, o clasă membră a unei alte clase, numită şi *clasă de acoperire*. In funcţie de situaţie, definirea unei clase interne se poate face fie ca membru al clasei de acoperire - caz în care este accesibilă tuturor metodelor, fie local în cadrul unei metode.

```
class ClasaDeAcoperire{
  class ClasaImbricata1 {
    // Clasa membru
  }
  void metoda() {
    class ClasaImbricata2 {
      // Clasa locala metodei
    }
  }
```

```
}
```

Folosirea claselor imbricate se face atunci când o clasă are nevoie în implementarea ei de o altă clasă şi nu există nici un motiv pentru care aceasta din urmă să fie declarată de sine stătătoare (nu mai este folosită nicăieri).

O clasă imbricată are un privilegiu special faţă de celelalte clase şi anume acces nerestricţionat la **toate** variabilele clasei de acoperire, chiar dacă acestea sunt private. O clasă declarată locală unei metode va avea acces şi la variabilele **finale** declarate în metoda respectivă.

```
class ClasaDeAcoperire{
  private int x=1;

  class ClasaImbricata1 {
    int a=x;
  }
  void metoda() {
    final int y=2;
    int z=3;
    class ClasaImbricata2 {
      int b=x;
      int c=y;

      int d=z; // Incorect
    }
  }
}
```

O clasă imbricată membră (care nu este locală unei metode) poate fi referită din exteriorul clasei de acoperire folosind expresia

`ClasaDeAcoperire.ClasaImbricata`

Aşadar, clasele membru pot fi declarate cu modificatorii `public, protected, private` pentru a controla nivelul lor de acces din exterior, întocmai ca orice variabilă sau metodă mebră a clasei. Pentru clasele imbricate locale unei metode nu sunt permişi acesţi modificatori.

Toate clasele imbricate pot fi declarate folosind modificatorii `abstract` şi `final`, semnificaţia lor fiind aceeaşi ca şi în cazul claselor obişnuite.

2.6.2 Clase interne

Spre deosebire de clasele obişnuite, o clasă imbricată poate fi declarată statică sau nu. O clasă imbricată nestatică se numeşte *clasa internă*.

```
class ClasaDeAcoperire{
  ...
  class ClasaInterna {
    ...
  }
  static class ClasaImbricataStatica {
    ...
  }
}
```

Diferenţierea acestor denumiri se face deoarece:

- o "clasă imbricată" reflectă relaţia sintactică a două clase: codul unei clase apare în interiorul codului altei clase;
- o "clasă internă" reflectă relaţia dintre instanţele a două clase, în sensul că o instanţa a unei clase interne nu poate exista decat în cadrul unei instanţe a clasei de acoperire.

In general, cele mai folosite clase imbricate sunt cele interne.

Aşadar, o clasă internă este o clasă imbricată ale carei instanţe nu pot exista decât în cadrul instanţelor clasei de acoperire şi care are acces direct la toţi membrii clasei sale de acoperire.

2.6.3 Identificare claselor imbricate

După cum ştim orice clasă produce la compilare aşa numitele "unităţi de compilare", care sunt fişiere având numele clasei respective şi extensia `.class` şi care conţin toate informaţiile despre clasa respectivă. Pentru clasele imbricate aceste unităţi de compilare sunt denumite astfel: numele clasei de acoperire, urmat de simbolul '$' apoi de numele clasei imbricate.

```
class ClasaDeAcoperire{
  class ClasaInterna1 {}
  class ClasaInterna2 {}
}
```

Pentru exemplul de mai sus vor fi generate trei fişiere:

```
ClasaDeAcoperire.class
ClasaDeAcoperire$ClasaInterna1.class
ClasaDeAcoperire$ClasaInterna2.class
```

In cazul în care clasele imbricate au la rândul lor alte clase imbricate (situaţie mai puţin uzuală) denumirea lor se face după aceeaşi regulă: adăugarea unui '$' şi apoi numele clasei imbricate.

2.6.4 Clase anonime

Există posibilitatea definirii unor clase imbricate locale, fără nume, utilizate doar pentru instanţierea unui obiect de un anumit tip. Astfel de clase se numesc *clase anonime* şi sunt foarte utile în situaţii cum ar fi crearea unor obiecte ce implementează o anumită interfaţă sau extind o anumită clasă abstractă.

Exemple de folosire a claselor anonime vor fi date în capitolul "Interfeţe", precum şi extensiv în capitolul "Interfaţa grafică cu utilizatorul".

Fişierele rezultate în urma compilării claselor anonime vor avea numele de forma `ClasaAcoperire.$1`,..., `ClasaAcoperire.$n`, unde n este numărul de clase anonime definite în clasa respectivă de acoperire.

2.7 Clase şi metode abstracte

Uneori în proiectarea unei aplicaţii este necesar să reprezentăm cu ajutorul claselor concepte abstracte care să nu poată fi instanţiate şi care să folosească doar la dezvoltarea ulterioară a unor clase ce descriu obiecte concrete. De exemplu, în pachetul `java.lang` există clasa abstractă `Number` care modelează conceptul generic de "număr". Intr-un program nu avem însă nevoie de numere generice ci de numere de un anumit tip: întregi, reale, etc. Clasa `Number` serveşte ca superclasă pentru clasele concrete `Byte`, `Double`, `Float`, `Integer`, `Long` şi `Short`, ce implementează obiecte pentru descrierea numerelor de un anumit tip. Aşadar, clasa `Number` reprezintă un concept abstract şi nu vom putea instanţia obiecte de acest tip - vom folosi în schimb subclasele sale.

```
Number  numar  = new Number();    // Eroare
Integer intreg = new Integer(10); // Corect
```

2.7.1 Declararea unei clase abstracte

Declararea unei clase abstracte se face folosind cuvântul rezervat `abstract`:

```
[public] abstract class ClasaAbstracta
   [extends Superclasa]
   [implements Interfata1, Interfata2, ...] {

   // Declaratii uzuale
   // Declaratii de metode abstracte
}
```

O clasă abstractă poate avea modificatorul `public`, accesul implicit fiind la nivel de pachet, dar nu poate specifica modificatorul `final`, combinaţia `abstract final` fiind semnalată ca eroare la compilare - de altfel, o clasă declarată astfel nu ar avea nici o utilitate.

O clasă abstractă poate conţine aceleaşi elemente membre ca o clasă obişnuită, la care se adaugă declaraţii de metode abstracte - fără nici o implementare.

2.7.2 Metode abstracte

Spre deosebire de clasele obişnuite care trebuie să furnizeze implementări pentru toate metodele declarate, o clasă abstractă poate conţine metode fără nici o implementare. Metodele fara nici o implementare se numesc *metode abstracte* şi pot apărea doar în clase abstracte. In faţa unei metode abstracte trebuie să apară obligatoriu cuvântul cheie `abstract`, altfel va fi furnizată o eroare de compilare.

```
abstract class ClasaAbstracta {
  abstract void metodaAbstracta(); // Corect
  void metoda();                   // Eroare
}
```

In felul acesta, o clasă abstractă poate pune la dispoziţia subclaselor sale un model complet pe care trebuie să-l implementeze, furnizând chiar implementarea unor metode comune tuturor claselor şi lăsând explicitarea altora

fiecărei subclase în parte.
Un exemplu elocvent de folosire a claselor şi metodelor abstracte este descrierea obiectelor grafice într-o manieră orientată-obiect.

- Obiecte grafice: linii, dreptunghiuri, cercuri, curbe Bezier, etc
- Stări comune: poziţia(originea), dimensiunea, culoarea, etc
- Comportament: mutare, redimensionare, desenare, colorare, etc.

Pentru a folosi stările şi comportamentele comune acestor obiecte în avantajul nostru putem declara o clasă generică `GraphicObject` care să fie superclasă pentru celelalte clase. Metodele abstracte vor fi folosite pentru implementarea comportamentului specific fiecărui obiect, cum ar fi desenarea iar cele obişnuite pentru comportamentul comun tuturor, cum ar fi schimbarea originii. Implementarea clasei abstracte `GraphicObject` ar putea arăta astfel:

```
abstract class GraphicObject {
  // Stari comune
  private int x, y;
  private Color color = Color.black;
  ...

  // Metode comune
  public void setX(int x) {
    this.x = x;
  }
  public void setY(int y) {
    this.y = y;
  }
  public void setColor(Color color) {
    this.color = color;
  }
  ...

  // Metode abstracte
  abstract void draw();
  ...
}
```

O subclasă care nu este abstractă a unei clase abstracte trebuie să furnizeze obligatoriu implementări ale metodelor abstracte definite în superclasă. Implementarea claselor pentru obiecte grafice ar fi:

```
class Circle extends GraphicObject {
  void draw() {
    // Obligatoriu implementarea
    ...
  }
}
class Rectangle extends GraphicObject {
  void draw() {
    // Obligatoriu implementarea
    ...
  }
}
```

Legat de metodele abstracte, mai trebuie menţionate următoarele:

- O clasă abstractă poate să nu aibă nici o metodă abstractă.
- O metodă abstractă nu poate apărea decât într-o clasă abstractă.
- Orice clasă care are o metodă abstractă trebuie declarată ca fiind abstractă.

In API-ul oferit de platforma de lucru Java sunt numeroase exemple de ierarhii care folosesc la nivelele superioare clase abstracte. Dintre cele mai importante amintim:

- `Number`: superclasa abstractă a tipurilor referinţă numerice
- `Reader`, `Writer`: superclasele abstracte ale fluxurilor de intrare/ieşire pe caractere
- `InputStream`, `OutputStream`: superclasele abstracte ale fluxurilor de intrare/ieşire pe octeţi
- `AbstractList`, `AbstractSet`, `AbstractMap`: superclase abstracte pentru structuri de date de tip colecţie

- `Component` : superclasa abstractă a componentelor folosite în dezvoltarea de aplicaţii cu interfaţă grafică cu utilizatorul (GUI), cum ar fi `Frame`, `Button`, `Label`, etc.

- etc.

2.8 Clasa Object

2.8.1 Orice clasă are o superclasă

După cum am văzut în secţiunea dedicată modalităţii de creare a unei clase, clauza "extends" specifică faptul că acea clasă extinde (moşteneşte) o altă clasă, numită superclasă. O clasă poate avea o singură superclasă (Java nu suportă moştenirea multiplă) şi chiar dacă nu specificăm clauza "extends" la crearea unei clase ea totuşi va avea o superclasă. Cu alte cuvinte, în Java orice clasă are o superclasă şi numai una. Evident, trebuie să existe o excepţie de la această regulă şi anume clasa care reprezintă rădăcina ierarhiei formată de relaţiile de moştenire dintre clase. Aceasta este clasa **Object**.

Clasa `Object` este şi superclasa implicită a claselor care nu specifică o anumită superclasă. Declaraţiile de mai jos sunt echivalente:

```
class Exemplu {}
class Exemplu extends Object {}
```

2.8.2 Clasa Object

Clasa `Object` este cea mai generală dintre clase, orice obiect fiind, direct sau indirect, descendent al acestei clase. Fiind părintele tuturor, `Object` defineşte şi implementează comportamentul comun al tuturor celorlalte clase Java, cum ar fi:

- posibilitatea testării egalităţii valorilor obiectelor,

- specificarea unei reprezentări ca şir de caractere a unui obiect ,

- returnarea clasei din care face parte un obiect,

- notificarea altor obiecte că o variabilă de condiţie s-a schimbat, etc.

Fiind subclasă a lui `Object`, orice clasă îi poate supradefini metodele care nu sunt finale. Metodele cel mai uzual supradefinite sunt: `clone`, `equals/hashCode`, `finalize`, `toString`.

- **clone**
  Această metodă este folosită pentru duplicarea obiectelor (crearea unor clone). Clonarea unui obiect presupune crearea unui nou obiect de acelaşi tip şi care să aibă aceeaşi stare (aceleaşi valori pentru variabilele sale).

- **equals**, **hashCode**
  Acestea sunt, de obicei, supradefinite împreună. In metoda `equals` este scris codul pentru compararea egalităţii conţinutului a două obiecte. Implicit (implementarea din clasa `Object`), această metodă compară referinţele obiectelor. Uzual este redefinită pentru a testa dacă stările obiectelor coincid sau dacă doar o parte din variabilele lor coincid.

  Metoda `hashCode` returneaza un cod întreg pentru fiecare obiect, pentru a testa consistenţa obiectelor: acelaşi obiect trebuie să returneze acelaşi cod pe durata execuţiei programului.
  Dacă două obiecte sunt egale conform metodei `equals`, atunci apelul metodei `hashCode` pentru fiecare din cele două obiecte ar trebui să returneze acelaşi intreg.

- **finalize**
  In această metodă se scrie codul care "curăţă după un obiect" înainte de a fi eliminat din memorie de colectorul de gunoaie. (vezi "Distrugerea obiectelor")

- **toString**
  Este folosită pentru a returna o reprezentare ca şir de caractere a unui obiect. Este utilă pentru concatenarea şirurilor cu diverse obiecte în vederea afişării, fiind apelată automat atunci când este necesară transformarea unui obiect în şir de caractere.

```
Exemplu obj = new Exemplu();
System.out.println("Obiect=" + obj);
//echivalent cu
System.out.println("Obiect=" + obj.toString());
```

Să considerăm următorul exemplu, în care implementăm parţial clasa numerelor complexe, şi în care vom supradefini metode ale clasei `Object`. De asemenea, vom scrie un mic program `TestComplex` în care vom testa metodele clasei definite.

Listing 2.1: Clasa numerelor complexe

```
class Complex {
  private double a; //partea reala
  private double b; //partea imaginara

  public Complex(double a, double b) {
    this.a = a;
    this.b = b;
  }

  public Complex() {
    this(1, 0);
  }

  public boolean equals(Object obj) {
    if (obj == null) return false;
    if (!(obj instanceof Complex)) return false;

    Complex comp = (Complex) obj;
    return ( comp.a==a && comp.b==b);
  }

  public Object clone() {
    return new Complex(a, b);
  }

  public String toString() {
    String semn = (b > 0 ? "+" : "-");
    return a + semn + b + "i";
  }

  public Complex aduna(Complex comp) {
    Complex suma = new Complex(0, 0);
    suma.a = this.a + comp.a;
    suma.b = this.b + comp.b;
    return suma;
  }
}
```

```
public class TestComplex {
  public static void main(String c[]) {
   Complex c1 = new Complex(1,2);
   Complex c2 = new Complex(2,3);
   Complex c3 = (Complex) c1.clone();
   System.out.println(c1.aduna(c2));  // 3.0 + 5.0i
   System.out.println(c1.equals(c2)); // false
   System.out.println(c1.equals(c3)); // true
  }
}
```

2.9 Conversii automate între tipuri

După cum văzut tipurile Java de date pot fi împărţie în *primitive* şi *referinţă*. Pentru fiecare tip primitiv există o clasă corespunzătoare care permie lucrul orientat obiect cu tipul respectiv.

byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Fiecare din aceste clase are un constructor ce permite iniţializarea unui obiect având o anumită valoare primitivă şi metode specializate pentru conversia unui obiect în tipul primitiv corespunzător, de genul *tipPrimitiv*`Value`:

```
Integer obi = new Integer(1);
int i = obi.intValue();

Boolean obb = new Boolean(true);
boolean b = obb.booleanValue();
```

Incepând cu versiunea 1.5 a limbajului Java, atribuirile explicite între tipuri primitve şi referinţă sunt posibile, acest mecanism purtând numele de *autoboxing*, respectiv *auto-unboxing*. Conversia explicită va fi facută de către compilator.

```
// Doar de la versiunea 1.5 !
Integer obi = 1;
int i = obi;

Boolean obb = true;
boolean b = obb;
```

2.10 Tipul de date enumerare

Incepând cu versiunea 1.5 a limbajului Java, există posibilitatea de a defini tipuri de date enumerare prin folosirea cuvântului cheie **enum**. Acestă soluţie simplifică manevrarea grupurilor de constante, după cum reiese din următorul exemplu:

```
public class CuloriSemafor {
  public static final int ROSU = -1;
  public static final int GALBEN = 0;
  public static final int VERDE = 1;
}
...
// Exemplu de utilizare
if (semafor.culoare = CuloriSemafor.ROSU)
  semafor.culoare = CuloriSemafor.GALBEN);
...
```

Clasa de mai sus poate fi rescrisă astfel:

```
public enum CuloriSemafor { ROSU, GALBEN, VERDE };
...
// Utilizarea structurii se face la fel
...
if (semafor.culoare = CuloriSemafor.ROSU)
  semafor.culoare = CuloriSemafor.GALBEN);
...
```

Compilatorul este responsabil cu transformarea unei astfel de structuri într-o clasă corespunzătoare.

Capitolul 3

Excepţii

3.1 Ce sunt excepţiile ?

Termenul *excepţie* este o prescurtare pentru "eveniment excepţional" şi poate fi definit ca un eveniment ce se produce în timpul execuţiei unui program şi care provoacă întreruperea cursului normal al execuţiei acestuia.

Excepţiile pot apărea din diverse cauze şi pot avea nivele diferite de gravitate: de la erori fatale cauzate de echipamentul hardware până la erori ce ţin strict de codul programului, cum ar fi accesarea unui element din afara spaţiului alocat unui vector.

In momentul când o asemenea eroare se produce în timpul execuţiei va fi generat un obiect de tip excepţie ce conţine:

- informaţii despre excepţia respectivă;
- starea programului în momentul producerii acelei excepţii.

```
public class Exemplu {
  public static void main(String args[]) {
    int v[] = new int[10];
    v[10] = 0;  //Exceptie !
    System.out.println("Aici nu se mai ajunge...");
  }
}
```

La rularea programului va fi generată o excepţie, programul se va opri la instrucţiunea care a cauzat excepţia şi se va afişa un mesaj de eroare de genul:

```
"Exception in thread "main"
  java.lang.ArrayIndexOutOfBoundsException :10
  at Exceptii.main (Exceptii.java:4)"
```

Crearea unui obiect de tip excepţie se numeşte *aruncarea unei excepţii* (”throwing an exception”). In momentul în care o metodă generează (aruncă) o excepţie sistemul de execuţie este responsabil cu găsirea unei secvenţe de cod dintr-o metodă care să o trateze. Căutarea se face recursiv, începând cu metoda care a generat excepţia şi mergând înapoi pe linia apelurilor către acea metodă.

Secvenţa de cod dintr-o metodă care tratează o anumită excepţie se numeşte *analizor de excepţie* (”exception handler”) iar interceptarea şi tratarea ei se numeşte *prinderea excepţiei* (”catch the exception”).
Cu alte cuvinte, la apariţia unei erori este ”aruncată” o excepţie iar cineva trebuie să o ”prindă” pentru a o trata. Dacă sistemul nu găseşte nici un analizor pentru o anumită excepţie, atunci programul Java se opreşte cu un mesaj de eroare (în cazul exemplului de mai sus mesajul ”Aici nu se mai ajunge...” nu va fi afişat).

Atenţie

In Java tratarea erorilor nu mai este o opţiune ci o constrângere. In aproape toate situaţile, o secvenţă de cod care poate provoca excepţii trebuie să specifice modalitatea de tratare a acestora.

3.2 ”Prinderea” şi tratarea excepţiilor

Tratarea excepţiilor se realizează prin intermediul blocurilor de instrucţiuni `try, catch` şi `finally`. O secvenţă de cod care tratează anumite excepţii trebuie să arate astfel:

```
try {
  // Instructiuni care pot genera exceptii
}
catch (TipExceptie1 variabila) {
  // Tratarea exceptiilor de tipul 1
```

```
}
catch (TipExceptie2 variabila) {
  // Tratarea exceptiilor de tipul 2
}
. . .
finally {
  // Cod care se executa indiferent
  // daca apar sau nu exceptii
}
```

Să considerăm următorul exemplu: citirea unui fişier octet cu octet şi afisarea lui pe ecran. Fără a folosi tratarea excepţiilor metoda responsabilă cu citirea fişierului ar arăta astfel:

```
public static void citesteFisier(String fis) {
  FileReader f = null;
  // Deschidem fisierul
  System.out.println("Deschidem fisierul " + fis);
  f = new FileReader(fis);

  // Citim si afisam fisierul caracter cu caracter
  int c;
  while ( (c=f.read()) != -1)
    System.out.print((char)c);

  // Inchidem fisierul
  System.out.println("\\nInchidem fisierul " + fis);
  f.close();
}
```

Această secvenţă de cod va furniza erori la compilare deoarece în Java tratarea erorilor este obligatorie. Folosind mecanismul excepţiilor metoda `citeste` îşi poate trata singură erorile care pot surveni pe parcursul execuţiei sale. Mai jos este codul complte şi corect al unui program ce afişează pe ecran conţinutul unui fişier al cărui nume este primit ca argument de la linia de comandă. Tratarea excepţiilor este realizată complet chiar de către metoda `citeste`.

Listing 3.1: Citirea unui fisier - corect

```
import java.io.*;

public class CitireFisier {
  public static void citesteFisier(String fis) {
    FileReader f = null;
    try {
      // Deschidem fisierul
      System.out.println("Deschidem fisierul " + fis);
      f = new FileReader(fis);

      // Citim si afisam fisierul caracter cu caracter
      int c;
      while ( (c=f.read()) != -1)
        System.out.print((char)c);

    } catch (FileNotFoundException e) {
      //Tratam un tip de exceptie
      System.err.println("Fisierul nu a fost gasit !");
      System.err.println("Exceptie: " + e.getMessage());
      System.exit(1);

    } catch (IOException e) {
      //Tratam alt tip de exceptie
      System.out.println("Eroare la citirea din fisier!");
      e.printStackTrace();

    } finally {
      if (f != null) {
        // Inchidem fisierul
        System.out.println("\nInchidem fisierul.");
        try {
           f.close();
        } catch (IOException e) {
          System.err.println("Fisierul nu poate fi inchis!");
          e.printStackTrace();
        }
      }
    }
  }

  public static void main(String args[]) {
    if (args.length > 0)
      citesteFisier(args[0]);
    else
```

```
        System.out.println("Lipseste numele fisierului!");
    }
}
```

Blocul "try" contine instrucţiunile de deschidere a unui fişier şi de citire dintr-un fişier, ambele putând produce excepţii. Excepţiile provocate de aceste instrucţiuni sunt tratate în cele două blocuri "catch", câte unul pentru fiecare tip de excepţie. Inchiderea fişierului se face în blocul "finally", deoarece acesta este sigur că se va executa Fără a folosi blocul "finally", închiderea fişierului ar fi trebuit facută în fiecare situaţie în care fişierul ar fi fost deschis, ceea ce ar fi dus la scrierea de cod redundant.

```
try {
  ...
  // Totul a decurs bine.
  f.close();
}
...
catch (IOException e) {
  ...
  // A aparut o exceptie la citirea din fisier
  f.close(); // cod redundant
}
```

O problemă mai delicată care trebuie semnalata în aceasta situaţie este faptul că metoda `close`, responsabilă cu închiderea unui fişier, poate provoca la rândul său excepţii, de exemplu atunci când fişierul mai este folosit şi de alt proces şi nu poate fi închis. Deci, pentru a avea un cod complet corect trebuie să tratăm şi posibilitatea apariţiei unei excepţii la metoda `close`.

Atenţie

Obligatoriu un bloc de instrucţiuni "try" trebuie să fie urmat de unul sau mai multe blocuri "catch", în funcţie de excepţiile provocate de acele instrucţiuni sau (opţional) de un bloc "finally".

3.3 "Aruncarea" excepţiilor

In cazul în care o metodă nu îşi asumă responsabilitatea tratării uneia sau mai multor excepţii pe care le pot provoca anumite instrucţiuni din codul său atunci ea poate să "arunce" aceste excepţii către metodele care o apelează, urmând ca acestea să implementeze tratarea lor sau, la rândul lor, să "arunce" mai departe excepţiile respective.
Acest lucru se realizează prin specificarea în declaraţia metodei a clauzei `throws`:

```
[modificatori] TipReturnat metoda([argumente])
    throws TipExceptie1, TipExceptie2, ...
{
  ...
}
```

Atenţie

O metodă care nu tratează o anumită excepţie trebuie obligatoriu să o "arunce".

In exemplul de mai sus dacă nu facem tratarea excepţiilor în cadrul metodei `citeste` atunci metoda apelantă (`main`) va trebui să facă acest lucru:

Listing 3.2: Citirea unui fisier

```
import java.io.*;

public class CitireFisier {
  public static void citesteFisier(String fis)
      throws FileNotFoundException, IOException {
    FileReader f = null;
    f = new FileReader(fis);

    int c;
    while ( (c=f.read()) != -1)
      System.out.print((char)c);

    f.close();
  }
```

```
  public static void main(String args[]) {
    if (args.length > 0) {
      try {
        citesteFisier(args[0]);

      } catch (FileNotFoundException e) {
        System.err.println("Fisierul nu a fost gasit !");
        System.err.println("Exceptie: " + e);

      } catch (IOException e) {
        System.out.println("Eroare la citirea din fisier!");
        e.printStackTrace();
      }

    } else
      System.out.println("Lipseste numele fisierului!");
  }
}
```

Observaţi că, în acest caz, nu mai putem diferenţia excepţiile provocate de citirea din fişier s de inchiderea fişierului, ambele fiind de tipul `IOException`. De asemenea, inchiderea fişierului nu va mai fi facută în situatia în care apare o excepţie la citirea din fişier. Este situaţia în care putem folosi blocul `finally` fără a folosi nici un bloc `catch`:

```
  public static void citesteFisier(String fis)
       throws FileNotFoundException, IOException {
    FileReader f = null;
    try {
      f = new FileReader(numeFisier);
      int c;
      while ( (c=f.read()) != -1)
        System.out.print((char)c);
    }
    finally {
      if (f!=null)
        f.close();
    }
  }
```

Metoda apelantă poate arunca la rândul său excepţiile mai departe către metoda care a apelat-o la rândul ei. Această înlănţuire se termină cu metoda `main` care, dacă va arunca excepţiile ce pot apărea în corpul ei, va determina trimiterea excepţiilor către maşina virtuală Java.

```
public void metoda3 throws TipExceptie {
  ...
}
public void metoda2 throws TipExceptie {
  metoda3();
}
public void metoda1 throws TipExceptie {
  metoda2();
}
public void main throws TipExceptie {
  metoda1();
}
```

Tratarea excepţiilor de către JVM se face prin terminarea programului şi afişarea informaţiilor despre excepţia care a determinat acest lucru. Pentru exemplul nostru, metoda `main` ar putea fi declarată astfel:

```
public static void main(String args[])
     throws FileNotFoundException, IOException {
  citeste(args[0]);
}
```

Intotdeauna trebuie găsit compromisul optim între tratarea locală a excepţiilor şi aruncarea lor către nivelele superioare, astfel încât codul să fie cât mai clar şi identificarea locului în care a apărut excepţia să fie cât mai uşor de făcut.

Aruncarea unei excepţii se poate face şi implicit prin instrucţiunea `throw` ce are formatul: `throw exceptie`, ca în exemplele de mai jos:

```
throw new IOException("Exceptie I/O");
...
if (index >= vector.length)
  throw new ArrayIndexOutOfBoundsException();
...
```

```
  catch(Exception e) {
    System.out.println("A aparut o exceptie);
    throw e;
  }
```

Această instrucţiune este folosită mai ales la aruncarea excepţiilor proprii. (vezi "Crearea propriilor excepţii")

3.4 Avantajele tratării excepţiilor

Prin modalitatea sa de tratare a excepţiilor, Java are următoarele avantaje faţă de mecanismul tradiţional de tratare a erorilor:

- Separarea codului pentru tratarea unei erori de codul în care ea poate să apară.
- Propagarea unei erori până la un analizor de excepţii corespunzător.
- Gruparea erorilor după tipul lor.

3.4.1 Separarea codului pentru tratarea erorilor

In programarea tradiţională tratarea erorilor se combină cu codul ce poate produce apariţia lor producând aşa numitul "cod spaghetti". Să considerăm următorul exemplu: o funcţie care încarcă un fişier în memorie:

```
  citesteFisier {
   deschide fisierul;
   determina dimensiunea fisierului;
   aloca memorie;
   citeste fisierul in memorie;
   inchide fisierul;
}
```

Problemele care pot apărea la aceasta funcţie, aparent simplă, sunt de genul: "Ce se întâmplă dacă: ... ?"

- fişierul nu poate fi deschis
- nu se poate determina dimensiunea fişierului

- nu poate fi alocată suficientă memorie
- nu se poate face citirea din fişier
- fişierul nu poate fi închis

Un cod tradiţional care să trateze aceste erori ar arăta astfel:

```
int citesteFisier() {
  int codEroare = 0;
  deschide fisierul;
  if (fisierul s-a deschis) {
    determina dimensiunea fisierului;
    if (s-a determinat dimensiunea) {
      aloca memorie;
      if (s-a alocat memorie) {
        citeste fisierul in memorie;
        if (nu se poate citi din fisier) {
          codEroare = -1;
        }
      } else {
         codEroare = -2;
      }
    } else {
      codEroare = -3;
    }
     inchide fisierul;
    if (fisierul nu s-a inchis && codEroare == 0) {
      codEroare = -4;
    } else {
      codEroare = codEroare & -4;
    }
  } else {
    codEroare = -5;
  }
  return codEroare;
} // Cod "spaghetti"
```

Acest stil de progamare este extrem de susceptibil la erori şi îngreunează extrem de mult înţtelegerea sa. In Java, folosind mecansimul excepţiilor, codul ar arata, schematizat, astfel:

```
int citesteFisier() {
   try {
    deschide fisierul;
    determina dimensiunea fisierului;
    aloca memorie;
    citeste fisierul in memorie;
    inchide fisierul;
  }
   catch (fisierul nu s-a deschis)
     {trateaza eroarea;}
   catch (nu s-a determinat dimensiunea)
     {trateaza eroarea;}
   catch (nu s-a alocat memorie)
     {trateaza eroarea}
   catch (nu se poate citi din fisier)
     {trateaza eroarea;}
   catch (nu se poate inchide fisierul)
     {trateaza eroarea;}
}
```

Diferenta de claritate este evidentă.

3.4.2 Propagarea erorilor

Propagarea unei erori se face până la un analizor de excepţii corespunzător. Să presupunem că apelul la metoda `citesteFisier` este consecinţa unor apeluri imbricate de metode:

```
int metoda1() {
  metoda2();
  ...
}
int metoda2() {
  metoda3;
  ...
}
int metoda3 {
  citesteFisier();
  ...
```

```
}
```

Să presupunem de asemenea că dorim să facem tratarea erorilor doar în `metoda1`. Tradiţional, acest lucru ar trebui făcut prin propagarea erorii produse de metoda `citesteFisier` până la `metoda1`:

```
int metoda1() {
  int codEroare = metoda2();
  if (codEroare != 0)
    //proceseazaEroare;
  ...
}
int metoda2() {
  int codEroare = metoda3();
  if (codEroare != 0)
    return codEroare;
  ...
}
int metoda3() {
  int codEroare = citesteFisier();
  if (codEroare != 0)
    return codEroare;
  ...
}
```

După cum am vazut, Java permite unei metode să arunce excepţiile apărute în cadrul ei la un nivel superior, adică funcţiilor care o apelează sau sistemului. Cu alte cuvinte, o metodă poate să nu îşi asume responsabilitatea tratării excepţiilor apărute în cadrul ei:

```
int metoda1() {
  try {
    metoda2();
  }
  catch (TipExceptie e) {
    //proceseazaEroare;
  }
  ...
}
```

```
int metoda2() throws TipExceptie {
  metoda3();
  ...
}
int metoda3() throws TipExceptie {
  citesteFisier();
  ...
}
```

3.4.3 Gruparea erorilor după tipul lor

In Java există clase corespunzătoare tuturor excepţiilor care pot apărea la execuţia unui program. Acestea sunt grupate în funcţie de similarităţile lor într-o ierarhie de clase. De exemplu, clasa `IOException` se ocupă cu excepţiile ce pot apărea la operaţii de intrare/iesire şi diferenţiază la rândul ei alte tipuri de excepţii, cum ar fi `FileNotFoundException`, `EOFException`, etc.
La rândul ei, clasa `IOException` se încadrează într-o categorie mai largă de excepţii şi anume clasa `Exception`.
Radacină acestei ierarhii este clasa `Throwable` (vezi "Ierarhia claselor ce descriu excepţii").

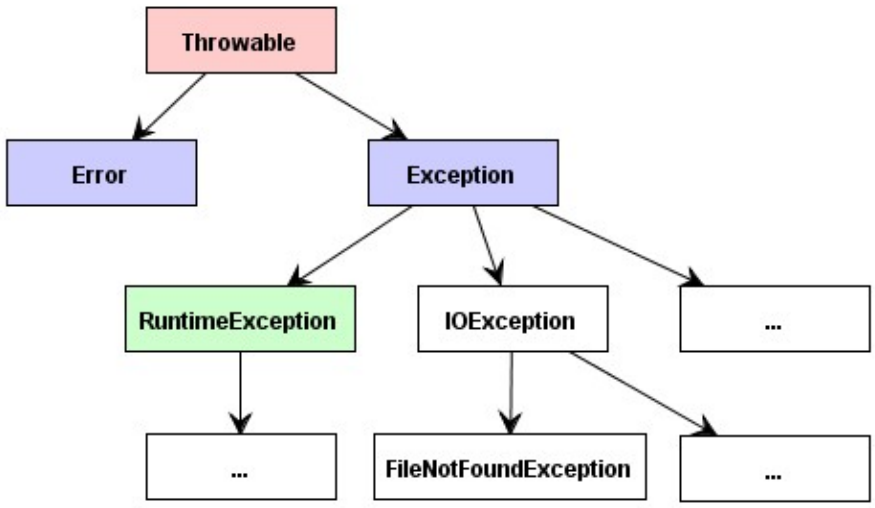
Pronderea unei excepţii se poate face fie la nivelul clasei specifice pentru acea excepţie, fie la nivelul uneia din superclasele sale, în funcţie de necesităţile programului, însă, cu cât clasa folosită este mai generică cu atât tratarea excepţiilor programul îşi pierde din flexibilitate.

```
try {
  FileReader f = new FileReader("input.dat");
  /* Acest apel poate genera exceptie
     de tipul FileNotFoundException
     Tratarea ei poate fi facuta in unul
     din modurile de mai jos:
  */
}
catch (FileNotFoundException e) {
  // Exceptie specifica provocata de absenta
  // fisierului 'input.dat'
} // sau
```

```
catch (IOException e) {
  // Exceptie generica provocata de o operatie IO
} // sau
catch (Exception e) {
  // Cea mai generica exceptie soft
} //sau
catch (Throwable e) {
  // Superclasa exceptiilor
}
```

3.5 Ierarhia claselor ce descriu excepţii

Rădăcina claselor ce descriu excepţii este clasa `Throwable` iar cele mai importante subclase ale sale sunt `Error, Exception` şi `RuntimeException`, care sunt la rândul lor superclase pentru o serie întreagă de tipuri de excepţii.



Erorile, obiecte de tip `Error`, sunt cazuri speciale de excepţii generate de funcţionarea anormală a echipamentului hard pe care rulează un program Java şi sunt invizibile programatorilor. Un program Java nu trebuie să trateze apariţia acestor erori şi este improbabil ca o metodă Java să provoace asemenea erori.

Excepţiile, obiectele de tip `Exception`, sunt excepţiile standard (soft) care trebuie tratate de către programele Java. După cum am mai zis tratarea acestor excepţii nu este o opţiune ci o constrângere. Excepţiile care pot "scăpa" netratate descind din subclasa `RuntimeException` şi se numesc *excepţii la execuţie*.

Metodele care sunt apelate uzual pentru un obiect excepţie sunt definite în clasa `Throwable` şi sunt publice, astfel încât pot fi apelate pentru orice tip de excepţie. Cele mai uzuale sunt:

- `getMessage` - afişează detaliul unei excepţii;
- `printStackTrace` - afişează informaţii complete despre excepţie şi localizarea ei;
- `toString` - metodă moştenită din clasa `Object`, care furnizează reprezentarea ca şir de caractere a excepţiei.

3.6 Excepţii la execuţie

In general, tratarea excepţiilor este obligatorie în Java. De la acest principu se sustrag însă aşa numitele *excepţii la execuţie* sau, cu alte cuvinte, excepţiile care provin strict din vina programatorului şi nu generate de o anumită situaţie externă, cum ar fi lipsa unui fişier.
Aceste excepţii au o superclasă comună `RuntimeException` şi în acesata categorie sunt incluse excepţiile provocate de:

- operaţii aritmetice ilegale (împârţirea întregilor la zero); `ArithmeticException`
- accesarea membrilor unui obiect ce are valoarea `null`; `NullPointerException`
- accesarea eronată a elementelor unui vector. `ArrayIndexOutOfBoundsException`

Excepţiile la execuţie pot apărea uriunde în program şi pot fi extrem de numeroare iar încercarea de "prindere" a lor ar fi extrem de anevoioasă. Din acest motiv, compilatorul permite ca aceste excepţii să rămână netratate, tratarea lor nefiind însă ilegală. Reamintim însă că, în cazul apariţiei oricărui tip de excepţie care nu are un analizor corespunzător, programul va fi terminat.

```
int v[] = new int[10];
try {
  v[10] = 0;
} catch (ArrayIndexOutOfBoundsException e) {
  System.out.println("Atentie la indecsi!");
  e.printStackTrace();
} // Corect, programul continua

v[11] = 0;
/* Nu apare eroare la compilare
   dar apare exceptie la executie si
   programul va fi terminat.
*/
System.out.println("Aici nu se mai ajunge...");
```

Impărţirea la 0 va genera o excepţie doar dacă tipul numerelor împărţite este aritmetic întreg. In cazul tipurilor reale (`float` şi `double`) nu va fi generată nici o excepţie, ci va fi furnizat ca rezultat o constantă care poate fi, funcţie de operatie, `Infinity`, `-Infinity`, sau `Nan`.

```
int a=1, int b=0;
System.out.println(a/b); // Exceptie la executie !

double x=1, y=-1, z=0;
System.out.println(x/z); // Infinity
System.out.println(y/z); // -Infinity
System.out.println(z/z); // NaN
```

3.7 Crearea propriilor excepţii

Adeseori poate apărea necesitatea creării unor excepţii proprii pentru a pune în evidenţa cazuri speciale de erori provocate de metodele claselor unei librării, cazuri care nu au fost prevazute în ierarhia excepţiilor standard Java.
O excepţie proprie trebuie să se încadreze însă în ierarhia excepţiilor Java, cu alte cuvinte clasa care o implementează trebuie să fie subclasă a uneia deja existente în aceasta ierarhie, preferabil una apropiată ca semnificaţie, sau superclasa `Exception`.

```
public class ExceptieProprie extends Exception {
  public ExceptieProprie(String mesaj) {
    super(mesaj);
    // Apeleaza constructorul superclasei Exception
  }
}
```

Să considerăm următorul exemplu, în care creăm o clasă ce descrie parţial o stivă de numere întregi cu operaţiile de adăugare a unui element, respectiv de scoatere a elementului din vârful stivei. Dacă presupunem că stiva poate memora maxim 100 de elemente, ambele operaţii pot provoca excepţii. Pentru a personaliza aceste excepţii vom crea o clasă specifică denumită `ExceptieStiva`:

Listing 3.3: Exceptii proprii

```
class ExceptieStiva extends Exception {
  public ExceptieStiva(String mesaj) {
    super(mesaj);
  }
}

class Stiva {

  int elemente[] = new int[100];
  int n=0; //numarul de elemente din stiva

  public void adauga(int x) throws ExceptieStiva {
    if (n==100)
      throw new ExceptieStiva("Stiva este plina!");
    elemente[n++] = x;
  }

  public int scoate() throws ExceptieStiva {
    if (n==0)
      throw new ExceptieStiva("Stiva este goala!");
    return elemente[n--];
  }
}
```

Secvenţa cheie este `extends Exception` care specifică faptul că noua clasă `ExceptieStiva` este subclasă a clasei `Exception` şi deci implementează obiecte ce reprezintă excepţii.

In general, codul adăugat claselor pentru excepţii proprii este nesemnificativ: unul sau doi constructori care afişeaza un mesaj de eroare la ieşirea standard. Procesul de creare a unei noi excepţii poate fi dus mai departe prin adăugarea unor noi metode clasei ce descrie acea excepţie, însă aceasta dezvoltare nu îşi are rostul în majoritatea cazurilor. Excepţiile proprii sunt descrise uzual de clase foarte simple, chiar fără nici un cod în ele, cum ar fi:

```
class ExceptieSimpla extends Exception { }
```

Această clasă se bazează pe constructorul implicit creat de compilator însă nu are constructorul `ExceptieSimpla(String s)`.

Capitolul 4

Intrări şi ieşiri

4.1 Introducere

4.1.1 Ce sunt fluxurile?

Majoritatea aplicaţiilor necesită citirea unor informaţii care se găsesc pe o sursă externă sau trimiterea unor informaţii către o destinaţie externă. Informaţia se poate găsi oriunde: într-un fişier pe disc, în reţea, în memorie sau în alt program şi poate fi de orice tip: date primitive, obiecte, imagini, sunete, etc. Pentru a aduce informaţii dintr-un mediu extern, un progam Java trebuie să deschidă un *canal de comunicaţie (flux)* de la sursa informaţiilor (fişier, memorie, socket, etc) şi să citească secvenţial informaţiile respective.

Similar, un program poate trimite informaţii către o destinaţie externă deschizând un canal de comunicaţie (flux) către acea destinaţie şi scriind secvenţial informaţiile respective.

Indiferent de tipul informaţiilor, citirea/scrierea de pe/către un mediu extern respectă următorul algoritm:

```
deschide canal comunicatie
while (mai sunt informatii) {
  citeste/scrie informatie;
}
inchide canal comunicatie;
```

Pentru a generaliza, atât sursa externă a unor date cât şi destinaţia lor sunt văzute ca fiind nişte procese care produc, respectiv consumă informaţii.

Definiţii:
Un *flux* este un canal de comunicaţie unidirecţional între două procese.
Un proces care descrie o sursă externă de date se numeşte *proces producător*.
Un proces care descrie o destinaţie externă pentru date se numeşte *proces consumator*.
Un flux care citeşte date se numeşte *flux de intrare*.
Un flux care scrie date se numeşte *flux de ieşire*.

Observaţii:
Fluxurile sunt canale de comunicaţie seriale pe 8 sau 16 biţi.
Fluxurile sunt unidirecţionale, de la producător la consumator.
Fiecare flux are un singur proces producător şi un singur proces consumator.
Intre două procese pot exista oricâte fluxuri, orice proces putând fi atât producator cât şi consumator în acelaşi timp, dar pe fluxuri diferite.
Consumatorul şi producatorul nu comunică direct printr-o interfaţă de flux ci prin intermediul codului Java de tratare a fluxurilor.

Clasele şi intefeţele standard pentru lucrul cu fluxuri se găsesc în pachetul **java.io**. Deci, orice program care necesită operaţii de intrare sau ieşire trebuie să conţină instrucţiunea de import a pachetului `java.io`:

```
import java.io.*;
```

4.1.2 Clasificarea fluxurilor

Există trei tipuri de clasificare a fluxurilor:

- După *direcţia* canalului de comunicaţie deschis fluxurile se împart în:
  - fluxuri de intrare (pentru citirea datelor)
  - fluxuri de ieşire (pentru scrierea datelor)
- După *tipul de date* pe care operează:
  - fluxuri de octeţi (comunicarea serială se realizează pe 8 biţi)
  - fluxuri de caractere (comunicarea serială se realizează pe 16 biţi)

- După *acţiunea* lor:
  - fluxuri primare de citire/scriere a datelor (se ocupă efectiv cu citirea/scrierea datelor)
  - fluxuri pentru procesarea datelor

4.1.3 Ierarhia claselor pentru lucrul cu fluxuri

Clasele rădăcină pentru ierarhiile ce reprezintă fluxuri de caractere sunt:

- **Reader**- pentru fluxuri de intrare şi
- **Writer**- pentru fluxuri de ieşire.

Acestea sunt superclase abstracte pentru toate clasele ce implementează fluxuri specializate pentru citirea/scrierea datelor pe 16 biţi şi vor conţine metodele comune tuturor.
Ca o regulă generală, toate clasele din aceste ierarhii vor avea terminaţia `Reader` sau `Writer` în funcţie de tipul lor, cum ar fi în exemplele: `FileReader`, `BufferedReader`, `FileWriter`, `BufferedWriter`, etc. De asemenea, se observă ca o altă regulă generală, faptul că unui flux de intrare *X*`Reader` îi corespunde uzual un flux de ieşire *X*`Writer`, însă acest lucru nu este obligatoriu.

Clasele radacină pentru ierarhia fluxurilor de octeţi sunt:

- **InputStream**- pentru fluxuri de intrare şi
- **OutputStream**- pentru fluxuri de ieşire.

Acestea sunt superclase abstracte pentru clase ce implementează fluxuri specializate pentru citirea/scrierea datelor pe 8 biţi. Ca şi în cazul fluxurilor pe caractere denumirile claselor vor avea terminaţia superclasei lor: `FileInputStream`, `BufferedInputStream`, `FileOutputStream`, `BufferedOutputStream`, etc., fiecărui flux de intrare *X*`InputStream` corespunzându-i uzual un flux de ieşire *X*`OutputStream`, fără ca acest lucru să fie obligatoriu.

Până la un punct, există un paralelism între ierarhia claselor pentru fluxuri de caractere şi cea pentru fluxurile pe octeţi. Pentru majoritatea programelor este recomandat ca scrierea şi citirea datelor să se facă prin intermediul fluxurilor de caractere, deoarece acestea permit manipularea caracterelor Unicode în timp ce fluxurile de octeţi permit doar lucrul pe 8 biti - caractere ASCII.

4.1.4 Metode comune fluxurilor

Superclasele abstracte `Reader` şi `InputStream` definesc metode similare pentru citirea datelor.

`Reader`	`InputStream`
`int read()`	`int read()`
`int read(char buf[])`	`int read(byte buf[])`
`...`	`...`

De asemenea, ambele clase pun la dispoziţie metode pentru marcarea unei locaţii într-un flux, saltul peste un număr de poziţii, resetarea poziţiei curente, etc. Acestea sunt însă mai rar folosite şi nu vor fi detaliate.

Superclasele abstracte `Writer` şi `OutputStream` sunt de asemenea paralele, definind metode similare pentru scrierea datelor:

`Reader`	`InputStream`
`void write(int c)`	`void write(int c)`
`void write(char buf[])`	`void write(byte buf[])`
`void write(String str)`	-
`...`	`...`

Inchiderea oricărui flux se realizează prin metoda **close**. In cazul în care aceasta nu este apelată explicit, fluxul va fi automat închis de către colectorul de gunoaie atunci când nu va mai exista nici o referinţă la el, însă acest lucru trebuie evitat deoarece, la lucrul cu fluxrui cu zonă tampon de memorie, datele din memorie vor fi pierdute la închiderea fluxului de către *gc*.

Metodele referitoare la fluxuri pot genera excepţii de tipul **IOException** sau derivate din această clasă, tratarea lor fiind obligatorie.

4.2 Folosirea fluxurilor

Aşa cum am văzut, fluxurile pot fi împărţite în funcţie de activitatea lor în fluxuri care se ocupă efectiv cu citirea/scrierea datelor şi fluxuri pentru procesarea datelor (de filtrare). In continuare, vom vedea care sunt cele mai importante clase din cele două categorii şi la ce folosesc acestea, precum şi modalităţile de creare şi utilizare a fluxurilor.

4.2.1 Fluxuri primitive

Fluxurile primitive sunt responsabile cu citirea/scrierea efectivă a datelor, punând la dispoziţie implementări ale metodelor de bază `read`, respectiv `write`, definite în superclase. In funcţie de tipul sursei datelor, ele pot fi împărţite astfel:

- **Fişier**
  `FileReader`, `FileWriter`
  `FileInputStream`, `FileOutputStream`
  Numite şi *fluxuri fişier*, acestea sunt folosite pentru citirea datelor dintr-un fişier, respectiv scrierea datelor într-un fişier şi vor fi analizate într-o secţiune separată (vezi "Fluxuri pentru lucrul cu fişiere").

- **Memorie**
  `CharArrayReader, CharArrayWriter`
  `ByteArrayInputStream`, `ByteArrayOutputStream`
  Aceste fluxuri folosesc pentru scrierea/citirea informaţiilor în/din memorie şi sunt create pe un vector existent deja. Cu alte cuvinte, permit tratarea vectorilor ca sursă/destinaţie pentru crearea unor fluxuri de intrare/ieşire.

  `StringReader, StringWriter`
  Permit tratarea şirurilor de caractere aflate în memorie ca sursă/destinaţie pentru crearea de fluxuri.

- **Pipe**
  `PipedReader, PipedWriter`
  `PipedInputStream, PipedOutputStream`
  Implementează componentele de intrare/ieşire ale unei conducte de

date (pipe). Pipe-urile sunt folosite pentru a canaliza ieşirea unui program sau fir de execuţie către intrarea altui program sau fir de execuţie.

4.2.2 Fluxuri de procesare

Fluxurile de procesare (sau de filtrare) sunt responsabile cu preluarea datelor de la un flux primitiv şi procesarea acestora pentru a le oferi într-o altă formă, mai utilă dintr-un anumit punct de vedere. De exemplu, `BufferedReader` poate prelua date de la un flux `FileReader` şi să ofere informaţia dintr-un fişier linie cu linie. Fiind primitiv, `FileReader` nu putea citi decât caracter cu caracter. Un flux de procesare nu poate fi folosit decât împreună cu un flux primitiv.

Clasele ce descriu aceste fluxuri pot fi împartite în funcţie de tipul de procesare pe care îl efectueaza astfel:

- **"Bufferizare"**
  `BufferedReader, BufferedWriter`
  `BufferedInputStream, BufferedOutputStream`
  Sunt folosite pentru a introduce un buffer în procesul de citire/scriere a informaţiilor, reducând astfel numărul de accesări la dispozitivul ce reprezintă sursa/destinaţia originală a datelor. Sunt mult mai eficiente decât fluxurile fără buffer şi din acest motiv se recomandă folosirea lor ori de câte ori este posibil (vezi "Citirea şi scrierea cu zona tampon").

- **Filtrare**
  `FilterReader, FilterWriter`
  `FilterInputStream, FilterOutputStream`
  Sunt clase abstracte ce definesc o interfaţă comună pentru fluxuri care filtrează automat datele citite sau scrise (vezi "Fluxuri pentru filtrare").

- **Conversie octeţi-caractere**
  `InputStreamReader, OutputStreamWriter`
  Formează o punte de legatură între fluxurile de caractere şi fluxurile de octeţi. Un flux `InputStreamReader` citeşte octeţi dintr-un flux `InputStream` şi îi converteşte la caractere, folosind codificarea standard a caracterelor sau o codificare specificată de program. Similar, un flux `OutputStreamWriter` converteşte caractere în octeţi şi trimite rezutatul către un flux de tipul `OutputStream`.

- **Concatenare**
  `SequenceInputStream`
  Concatenează mai multe fluxuri de intrare într-unul singur (vezi "Concatenarea fişierelor").

- **Serializare**
  `ObjectInputStream, ObjectOutputStream`
  Sunt folosite pentru serializarea obiectelor (vezi "Serializarea obiectelor").

- **Conversie tipuri de date**
  `DataInputStream, DataOutputStream`
  Folosite la scrierea/citirea datelor de tip primitiv într-un format binar, independent de maşina pe care se lucrează (vezi "Folosirea claselor DataInputStream şi DataOutputStream").

- **Numărare**
  `LineNumberReader`
  `LineNumberInputStream`
  Oferă şi posibilitatea de numărare automată a liniilor citite de la un flux de intrare.

- **Citire în avans**
  `PushbackReader`
  `PushbackInputStream`
  Sunt fluxuri de intrare care au un buffer de 1-caracter(octet) în care este citit în avans şi caracterul (octetul) care urmează celui curent citit.

- **Afişare**
  `PrintWriter`
  `PrintStream`
  Oferă metode convenabile pentru afisarea informaţiilor.

4.2.3 Crearea unui flux

Orice flux este un obiect al clasei ce implementează fluxul respectiv. Crearea unui flux se realizează aşadar similar cu crearea obiectelor, prin instrucţiunea **new** şi invocarea unui constructor corespunzător al clasei respective:

Exemple:

```
//crearea unui flux de intrare pe caractere
FileReader in = new FileReader("fisier.txt");

//crearea unui flux de iesire pe caractere
FileWriter out = new FileWriter("fisier.txt");

//crearea unui flux de intrare pe octeti
FileInputStream in = new FileInputStream("fisier.dat");

//crearea unui flux de iesire pe octeti
FileOutputStrem out = new FileOutputStream("fisier.dat");
```

Aşadar, crearea unui flux primitiv de date care citeşte/scrie informaţii de la un dispozitiv extern are formatul general:

FluxPrimitiv numeFlux = new FluxPrimitiv(dispozitivExtern);

Fluxurile de procesare nu pot exista de sine stătătoare ci se suprapun pe un flux primitiv de citire/scriere a datelor. Din acest motiv, constructorii claselor pentru fluxurile de procesare nu primesc ca argument un dispozitiv extern de memorare a datelor ci o referinţa la un flux primitiv responsabil cu citirea/scrierea efectivă a datelor:

```
Exemple:
  //crearea unui flux de intrare printr-un buffer
  BufferedReader in = new BufferedReader(
      new FileReader("fisier.txt"));
  //echivalent cu
  FileReader fr = new FileReader("fisier.txt");
  BufferedReader in = new BufferedReader(fr);

  //crearea unui flux de iesire printr-un buffer
  BufferedWriter out = new BufferedWriter(
      new FileWriter("fisier.txt")));
  //echivalent cu
  FileWriter fo = new FileWriter("fisier.txt");
  BufferedWriter out = new BufferedWriter(fo);
```

Aşadar, crearea unui flux pentru procesarea datelor are formatul general:

FluxProcesare numeFlux = new FluxProcesare(fluxPrimitiv);

In general, fluxurile pot fi compuse în succesiuni oricât de lungi:

```
DataInputStream in = new DataInputStream(
    new BufferedInputStream(
      new FileInputStream("fisier.dat")));
```

4.2.4 Fluxuri pentru lucrul cu fişiere

Fluxurile pentru lucrul cu fişiere sunt cele mai usor de înteles, întrucât operaţia lor de bază este citirea, respectiv scrierea unui caracter sau octet dintr-un sau într-un fişier specificat uzual prin numele său complet sau relativ la directorul curent.

După cum am văzut deja, clasele care implementează aceste fluxuri sunt următoarele:

```
FileReader, FileWriter            - caractere
FileInputStream, FileOutputStream - octeti
```

Constructorii acestor clase acceptă ca argument un obiect care să specifice un anume fişier. Acesta poate fi un şir de caractere, on obiect de tip `File` sau un obiect de tip `FileDesciptor` (vezi "Clasa File").

Constructorii clasei `FileReader` sunt:

```
public FileReader(String fileName)
    throws FileNotFoundException
public FileReader(File  file)
    throws FileNotFoundException
public FileReader(FileDescriptor fd)
```

Constructorii clasei `FileWriter`:

```
public FileWriter(String fileName)
    throws IOException
public FileWriter(File file)
    throws IOException
public FileWriter(FileDescriptor fd)
public FileWriter(String fileName, boolean  append)
    throws IOException
```

Cei mai uzuali constructori sunt cei care primesc ca argument numele fişierului. Aceştia pot provoca excepţii de tipul `FileNotFoundException` în cazul în care fişierul cu numele specificat nu există. Din acest motiv orice creare a unui flux de acest tip trebuie făcută într-un bloc `try-catch` sau metoda în care sunt create fluxurile respective trebuie să arunce excepţiile de tipul `FileNotFoundException` sau de tipul superclasei `IOException`.

Să considerăm ca exemplu un program care copie conţinutul unui fişier cu numele "in.txt" într-un alt fişier cu numele "out.txt". Ambele fişiere sunt considerate în directorul curent.

Listing 4.1: Copierea unui fisier

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

    try {
      FileReader in  = new FileReader("in.txt");
      FileWriter out = new FileWriter("out.txt");

      int c;
      while ((c = in.read()) != -1)
        out.write(c);

      in.close();
      out.close();

    } catch(IOException e) {
      System.err.println("Eroare la operatiile cu fisiere!");
      e.printStackTrace();
    }
  }
}
```

In cazul în care vom lansa aplicaţia iar în directorul curent nu există un fişier cu numele "in.txt", va fi generată o excepţie de tipul `FileNotFoundException`. Aceasta va fi prinsă de program deoarece, `IOException` este superclasă pentru `FileNotFoundException`.
Dacă există fişierul "in.txt", aplicaţia va crea un nou fişier "out.txt" în care va fi copiat conţinutul primului. Dacă există deja fişierul "out.txt" el va fi re-

scris. Dacă doream să facem operaţia de adăugare(append) şi nu de rescriere pentru fişierul "out.txt" foloseam:

```
FileWriter out = new FileWriter("out.txt", true);
```

4.2.5 Citirea şi scrierea cu buffer

Clasele pentru citirea/scrierea cu zona tampon sunt:

```
BufferedReader, BufferedWriter            - caractere
BufferedInputStream, BufferedOutputStream - octeti
```

Sunt folosite pentru a introduce un buffer (zonă de memorie) în procesul de citire/scriere a informaţiilor, reducând astfel numarul de accesări ale dispozitivului ce reprezintă sursa/destinaţia atelor. Din acest motiv, sunt mult mai eficiente decât fluxurile fără buffer şi din acest motiv se recomandă folosirea lor ori de câte ori este posibil.

Clasa `BufferedReader` citeşte în avans date şi le memorează într-o zonă tampon. Atunci când se execută o operaţie de citire, caracterul va fi preluat din buffer. In cazul în care buffer-ul este gol, citirea se face direct din flux şi, odată cu citirea caracterului, vor fi memorati în buffer şi caracterele care îi urmează. Evident, `BufferedInputStream` funcţionează după acelaşi principiu, singura diferenţă fiind faptul că sunt citiţi octeţi.

Similar lucreaza şi clasele `BufferedWriter` şi `BufferedOutputStream`. La operaţiile de scriere datele scrise nu vor ajunge direct la destinaţie, ci vor fi memorate jntr-un buffer de o anumită dimensiune. Atunci când bufferul este plin, conţinutul acestuia va fi transferat automat la destinaţie.

Fluxurile de citire/scriere cu buffer sunt fluxuri de procesare şi sunt folosite prin suprapunere cu alte fluxuri, dintre care obligatoriu unul este primitiv.

```
BufferedOutputStream out = new BufferedOutputStream(
  new FileOutputStream("out.dat"), 1024)
//1024 este dimensiunea bufferului
```

Constructorii cei mai folosiţi ai acestor clase sunt următorii:

```
BufferedReader(Reader in)
BufferedReader(Reader in, int dim_buffer)
BufferedWriter(Writer out)
```

```
BufferedWriter(Writer out, int dim_buffer)
BufferedInputStream(InputStream in)
BufferedInputStream(InputStream in, int dim_buffer)
BufferedOutputStream(OutputStream out)
BufferedOutputStream(OutputStream out, int dim_buffer)
```

In cazul constructorilor în care dimensiunea buffer-ului nu este specificată, aceasta primeşte valoarea implicită de 512 octeţi (caractere).

Metodele acestor clase sunt cele uzuale de tipul `read` şi `write`. Pe lânga acestea, clasele pentru scriere prin buffer mai au şi metoda `flush` care goleşte explicit zona tampon, chiar dacă aceasta nu este plină.

```
BufferedWriter out = new BufferedWriter(
  new FileWriter("out.dat"), 1024)
//am creat un flux cu buffer de 1024 octeti
for(int i=0; i<1000; i++)
  out.write(i);
//bufferul nu este plin, in fisier nu s-a scris nimic
out.flush();
//bufferul este golit, datele se scriu in fisier
```

Metoda readLine

Este specifică fluxurilor de citire cu buffer şi permite citirea linie cu linie a datelor de intrare. O linie reprezintă o succesiune de caractere terminată cu simbolul pentru sfârşit de linie, dependent de platforma de lucru. Acesta este reprezentat în Java prin secvenţa escape '$\backslash n$';

```
BufferedReader br = new BufferedReader(new FileReader("in"))
String linie;
while ((linie = br.readLine()) != null) {
  ...
  //proceseaza linie
}
br.close();
}
```

4.2.6 Concatenarea fluxurilor

Clasa **SequenceInputStream** permite unei aplicatii să combine serial mai multe fluxuri de intrare astfel încât acestea să apară ca un singur flux de intrare. Citirea datelor dintr-un astfel de flux se face astfel: se citeşte din primul flux de intrare specificat pâna când se ajunge la sfârsitul acestuia, după care primul flux de intrare este închis şi se deschide automat următorul flux de intrare din care se vor citi în continuare datele, după care procesul se repetă pâna la terminarea tuturor fluxurilor de intrare.

Constructorii acestei clase sunt:

```
SequenceInputStream(Enumeration e)
SequenceInputStream(InputStream s1, InputStream s2)
```

Primul construieste un flux secvential dintr-o mulţime de fluxuri de intrare. Fiecare obiect în enumerarea primită ca parametru trebuie să fie de tipul `InputStream`.
Cel de-al doilea construieşte un flux de intrare care combină doar două fluxuri $s1$ şi $s2$, primul flux citit fiind $s1$.

Exemplul cel mai elocvent de folosirea a acestei clase este concatenarea a două sau mai multor fişiere:

Listing 4.2: Concatenarea a două fişiere

```
/* Concatenarea a doua fisiere
   ale caror nume sunt primite de la linia de comanda.
   Rezultatul concatenarii este afisat pe ecran.
*/
import java.io.*;
public class Concatenare {
  public static void main(String args[]) {
    if (args.length <= 1) {
      System.out.println("Argumente insuficiente!");
      System.exit(-1);
    }
    try {
      FileInputStream f1 = new FileInputStream(args[0]);
      FileInputStream f2 = new FileInputStream(args[1]);
      SequenceInputStream s = new SequenceInputStream(f1, f2)
          ;
      int c;
      while ((c = s.read()) != -1)
        System.out.print((char)c);
```

```
      s.close();
      //f1 si f2 sunt inchise automat
    } catch (IOException e) {
        e.printStackTrace();
    }
  }
}
```

Pentru concatenarea mai multor fişiere există două variante:

- folosirea unei enumerări - primul constructor (vezi "Colecţii");
- concatenarea pe rând a acestora folosind al 2-lea constructor; concatenarea a 3 fişiere va construi un flux de intrare astfel:

```
FileInputStream f1 = new FileInputStream(args[0]);
FileInputStream f2 = new FileInputStream(args[1]);
FileInputStream f3 = new FileInputStream(args[2]);
SequenceInputStream s = new SequenceInputStream(
  f1, new SequenceInputStream(f2, f3));
```

4.2.7 Fluxuri pentru filtrarea datelor

Un flux de filtrare se ataşează altui flux pentru a filtra datele care sunt citite/scrise de către acel flux. Clasele pentru filtrarea datelor superclasele abstracte:

- **FilterInputStream** - pentru filtrarea fluxurilor de intrare şi
- **FilterOutputStream** - pentru filtrarea fluxurilor de ieşire.

Cele mai importante fluxruri pentru filtrarea datelor sunt implementate de clasele:

```
DataInputStream, DataOutputStream
BufferedInputStream, BufferedOutputStream
LineNumberInputStream
PushbackInputStream
PrintStream
```

Observaţi că toate aceste clase descriu fluxuri de octeţi.

Filtrarea datelor nu trebuie văzută ca o metodă de a elimina anumiţi octeti dintr-un flux ci de a transforma aceşti octeţi în date care să poată fi interpretate sub altă formă. Aşa cum am văzut la citirea/scrierea cu zonă tampon, clasele de filtrare `BufferedInputStream` şi `BufferedOutputStream` colectează datele unui flux într-un buffer, urmând ca citirea/scrierea să se facă prin intermediu acelui buffer.

Aşadar, fluxurile de filtrare nu elimină date citite sau scrise de un anumit flux, ci introduc o noua modalitate de manipulare a lor, ele mai fiind numite şi *fluxuri de procesare*. Din acest motiv, fluxurile de filtrare vor conţine anumite metode specializate pentru citirea/scrierea datelor, altele decât cele comune tuturor fluxurilor. De exemplu, clasa `BufferedInputStream` pune la dispoziţie metoda `readLine` pentru citirea unei linii din fluxul de intrare.

Folosirea fluxurilor de filtrare se face prin ataşarea lor de un flux care se ocupă efectiv de citirea/scrierea datelor:

```
FluxFiltrare numeFlux = new FluxFiltrare(referintaAltFlux);
```

4.2.8 Clasele `DataInputStream` şi `DataOutputStream`

Aceste clase oferă metode prin care un flux nu mai este văzut ca o însiruire de octeţi, ci de date primitive. Prin urmare, vor furniza metode pentru citirea şi scrierea datelor la nivel de tip primitiv şi nu la nivel de octet. Clasele care oferă un astfel de suport implementează interfeţele `DataInput`, respectiv `DataOutput`. Acestea definesc metodele pe care trebuie să le pună la dispoziţie în vederea citireii/scrierii datelor de tip primitiv. Cele mai folosite metode, altele decât cele comune tuturor fluxurilor, sunt date în tabelul de mai jos:

DataInputStream	DataOutputStream
readBoolean	writeBoolean
readByte	writeByte
readChar	writeChar
readDouble	writeDouble
readFloat	writeFloat
readInt	writeInt
readLong	writeLong
readShort	writeShort
readUTF	writeUTF

Aceste metode au denumirile generice de **read***XXX* şi **write***XXX*, specificate de interfetele `DataInput` şi `DataOutput` şi pot provoca excepţii de tipul `IOException`. Denumirile lor sunt sugestive pentru tipul de date pe care îl prelucrează. mai puţin `readUTF` şi writeUTF care se ocupă cu obiecte de tip `String`, fiind singurul tip referinţă permis de aceste clase.

Scrierea datelor folosind fluxuri de acest tip se face în format binar, ceea ce înseamnă că un fişier în care au fost scrise informaţii folosind metode `write`*XXX* nu va putea fi citit decât prin metode `read`*XXX*.

Transformarea unei valori în format binar se numeşte *serializare*. Clasele `DataInputStream` şi `DataOutputStream` permit serializarea tipurilor primitive şi a şirurilor de caractere. Serializarea celorlalte tipuri referinţă va fi făcută prin intermediul altor clase, cum ar fi `ObjectInputStream` şi `ObjectOutputStream` (vezi "Serializarea obiectelor").

4.3 Intrări şi ieşiri formatate

Incepând cu versiunea 1.5, limbajul Java pune la dispoziţii modalităţi simplificate pentru afişarea formatată a unor informaţii, respectiv pentru citirea de date formatate de la tastatură.

4.3.1 Intrări formatate

Clasa **java.util.Scanner** oferă o soluţie simplă pentru formatarea unor informaţii citite de pe un flux de intrare fie pe octeţi, fie pe caractere, sau chiar dintr-un obiect de tip `File`. Pentru a citi de la tastatură vom specifica ca argument al constructorului fluxul `System.in`:

```
Scanner s = Scanner.create(System.in);
String nume = s.next();
int varsta = s.nextInt();
double salariu = s.nextDouble();
s.close();
```

4.3.2 Ieşiri formatate

Clasele `PrintStream` şi `PrintWriter` pun la dispoziţiile, pe lângă metodele `print, println` care ofereau posibilitatea de a afişa un şir de caractere, şi metodele **format**, **printf** (echivalente) ce permit afişarea formatată a unor variabile.

```
System.out.printf("%s %8.2f %2d %n", nume, salariu, varsta);
```

Formatarea şirurilor de caractere se bazează pe clasa **java.util.Formatter**.

4.4 Fluxuri standard de intrare şi ieşire

Mergând pe linia introdusă de sistemul de operare UNIX, orice program Java are :

- o intrare standard
- o ieşire standard
- o ieşire standard pentru erori

In general, intrarea standard este tastatura iar ieşirea standard este ecranul.

Intrarea şi ieşirea standard sunt reprezentate de obiecte pre-create ce descriu fluxuri de date care comunică cu dispozitivele standard ale sistemului. Aceste obiecte sunt definite publice în clasa `System` şi sunt:

- **System.in** - fluxul standar de intrare, de tip `InputStream`
- **System.out** - fluxul standar de ieşire, de tip `PrintStream`
- **System.err** - fluxul standar pentru erori, de tip `PrintStream`

4.4.1 Afisarea informaţiilor pe ecran

Am văzut deja numeroase exemple de utilizare a fluxului standard de ieşire, el fiind folosit la afişarea oricăror rezultate pe ecran (în modul consola):

```
System.out.print  (argument);
System.out.println(argument);
System.out.printf (format, argumente...);
System.out.format (format, argumente...);
```

Fluxul standard pentru afişarea erorilor se foloseşte similar şi apare uzual în secvenţele de tratare a excepţiilor. Implicit, este acelaşi cu fluxul standard de ieşire.

```
catch(Exception e) {
  System.err.println("Exceptie:" + e);
}
```

Fluxurile de ieşire pot fi folosite aşadar fără probleme deoarece tipul lor este `PrintStream`, clasă concretă pentru scrierea datelor. In schimb, fluxul standard de intrare `System.out` este de tip `InputStream`, care este o clasă abstractă, deci pentru a-l putea utiliza eficient va trebui sa-l folosim împreuna cu un flux de procesare(filtrare) care să permită citirea facilă a datelor.

4.4.2 Citirea datelor de la tastatură

Uzual, vom dori să folosim metoda `readLine` pentru citirea datelor de la tastatura şi din acest motiv vom folosi intrarea standard împreună cu o clasă de procesare care oferă această metodă. Exemplul tipic este:

```
BufferedReader stdin = new BufferedReader(
  new InputStreamReader(System.in));
System.out.print("Introduceti o linie:");
String linie = stdin.readLine()
System.out.println(linie);
```

In exemplul următor este prezentat un program care afişeaza liniile introduse de la tastatură până în momentul în care se introduce linia "exit" sau o linie vidă şi menţionează dacă şirul respectiv reprezintă un număr sau nu.

Listing 4.3: Citirea datelor de la tastatură

```
/* Citeste siruri de la tastatura si verifica
   daca reprezinta numere sau nu
*/
import java.io.*;
public class EsteNumar {
public static void main(String[] args) {
  BufferedReader stdin = new BufferedReader(
    new InputStreamReader(System.in));
    try {
      while(true) {
        String s = stdin.readLine();
        if (s.equals("exit") || s.length()==0)
          break;
        System.out.print(s);
        try {
          Double.parseDouble(s);
          System.out.println(": DA");
        } catch(NumberFormatException e) {
          System.out.println(": NU");
        }
      }
    } catch(IOException e) {
      System.err.println("Eroare la intrarea standard!");
      e.printStackTrace();
    }
  }
}
```

Incepând cu versiunea 1.5, varianta cea mai comodă de citire a datelor de la tastatură este folosirea clasei `java.util.Scanner`.

4.4.3 Redirectarea fluxurilor standard

Redirectarea fluxurilor standard presupune stabilirea unei alte surse decât tastatura pentru citirea datelor, respectiv alte destinaţii decât ecranul pentru cele două fluxuri de ieşire. In clasa `System` există următoarele metode statice care realizează acest lucru:

```
setIn(InputStream)  - redirectare intrare
setOut(PrintStream) - redirectare iesire
setErr(PrintStream) - redirectare erori
```

Redirectarea ieşirii este utilă în special atunci când sunt afişate foarte multe date pe ecran. Putem redirecta afisarea către un fişier pe care să-l citim după execuţia programului. Secvenţa clasică de redirectare a ieşirii este către un fişier este:

```
PrintStream fis = new PrintStream(
   new FileOutputStream("rezultate.txt")));
System.setOut(fis);
```

Redirectarea erorilor într-un fişier poate fi de asemenea utilă şi se face într-o manieră similară:

```
PrintStream fis = new PrintStream(
   new FileOutputStream("erori.txt")));
System.setErr(fis);
```

Redirectarea intrării poate fi folositoare pentru un program în mod consolă care primeşte mai multe valori de intrare. Pentru a nu le scrie de la tastatură de fiecare dată în timpul testării programului, ele pot fi puse într-un fişier, redirectând intrarea standard către acel fişier. In momentul când testarea programului a luat sfârsit redirectarea poate fi eliminată, datele fiind cerute din nou de la tastatură.

Listing 4.4: Exemplu de folosire a redirectării:

```
import java.io.*;
class Redirectare {
  public static void main(String[] args) {
    try {
      BufferedInputStream in = new BufferedInputStream(
        new FileInputStream("intrare.txt"));
      PrintStream out = new PrintStream(
          new FileOutputStream("rezultate.txt"));
      PrintStream err = new PrintStream(
          new FileOutputStream("erori.txt"));

      System.setIn(in);
      System.setOut(out);
      System.setErr(err);

      BufferedReader br = new BufferedReader(
        new InputStreamReader(System.in));
```

```
      String s;
      while((s = br.readLine()) != null) {
        /* Liniile vor fi citite din fisierul intrare.txt
           si vor fi scrise in fisierul rezultate.txt
        */
        System.out.println(s);
      }

      //Aruncam fortat o exceptie
      throw new IOException("Test");

    } catch(IOException e) {
      /* Daca apar exceptii,
         ele vor fi scrise in fisierul erori.txt
      */
      System.err.println("Eroare intrare/iesire!");
      e.printStackTrace();
    }
  }
}
```

4.4.4 Analiza lexicală pe fluxuri (clasa `StreamTokenizer`)

Clasa `StreamTokenizer` procesează un flux de intrare de orice tip şi îl împarte în "atomi lexicali". Rezultatul va consta în faptul că în loc să se citească octeţi sau caractere, se vor citi, pe rând, atomii lexicali ai fluxului respectiv. Printr-un *atom lexical* se în]elege în general:

- un identificator (un şir care nu este între ghilimele)
- un numar
- un şir de caractere
- un comentariu
- un separator

Atomii lexicali sunt despărţiti între ei de separatori. Implicit, aceşti separatori sunt cei obişnuţi: spaţiu, tab, virgulă, punct şi virgula, etc., însă pot fi schimbaţi prin diverse metode ale clasei.

Constructorii clasei sunt:

```
public StreamTokenizer(Reader r)
public StreamTokenizer(InputStream is)
```

Identificarea tipului şi valorii unui atom lexical se face prin intermediul variabilelor:

- **TT_EOF** - atom ce marchează sfâarşitul fluxului
- **TT_EOL** - atom ce marchează sfârşitul unei linii
- **TT_NUMBER** - atom de tip număr
- **TT_WORD**- atom de tip cuvânt
- **ttype**- tipul ultimului atom citit din flux
- **nval**- valoarea unui atom numeric
- **sval** - valoarea unui atom de tip cuvânt

Citirea atomilor din flux se face cu metoda **nextToken()**, care returnează tipul atomului lexical citit şi scrie în variabilele **nval** sau **sval** valoarea corespunzatoăre atomului.

Exemplul tipic de folosire a unui analizor lexical este citirea unei secvenţe de numere şi şiruri aflate într-un fişier sau primite de la tastatură:

Listing 4.5: Citirea unor atomi lexicali dintr-un fisier

```
/* Citirea unei secvente de numere si siruri
   dintr-un fisier specificat
   si afisarea tipului si valorii lor
*/

import java.io.*;
public class CitireAtomi {
  public static void  main(String args[]) throws IOException{

    BufferedReader br = new BufferedReader(new FileReader("
       fisier.txt"));
    StreamTokenizer st = new StreamTokenizer(br);

    int tip = st.nextToken();
    //Se citeste primul atom lexical
```

```
    while (tip != StreamTokenizer.TT_EOF) {
        switch (tip) {
          case StreamTokenizer.TT_WORD:
            System.out.println("Cuvant: " + st.sval);
            break;
          case StreamTokenizer.TT_NUMBER:
            System.out.println("Numar: " + st.nval);
        }

        tip = st.nextToken();
        //Trecem la urmatorul atom
    }
  }
}
```

Aşadar, modul de utilizare tipic pentru un analizor lexical este într-o bucla "while", în care se citesc atomii unul câte unul cu metoda `nextToken`, pâna se ajunge la sfârsitul fluxului (`TT_EOF`). In cadrul buclei "while" se determină tipul atomul curent curent (întors de metoda `nextToken`) şi apoi se află valoarea numerică sau şirul de caractere corespunzător atomului respectiv.

In cazul în care tipul atomilor nu ne interesează, este mai simplu să citim fluxul linie cu linie şi să folosim clasa `StringTokenizer`, care realizează împărţirea unui şir de caractere în atomi lexicali, sau metoda `split` a clasei `String`.

4.5 Clasa `RandomAccesFile` (fişiere cu acces direct)

După cum am văzut, fluxurile sunt procese secvenţiale de intrare/ieşire. Acestea sunt adecvate pentru lucrul cu medii secvenţiale de memorare a datelor, cum ar fi banda magnetică sau pentru transmiterea informaţiilor prin reţea, desi sunt foarte utile şi pentru dispozitive în care informaţia poate fi accesată direct.

Clasa `RandomAccesFile` are următoarele caracteristici:

- permite accesul nesecvenţial (direct) la conţinutul unui fişier;
- este o clasă de sine stătătoare, subclasă directă a clasei `Object`;
- se găseşte în pachetul `java.io`;

- implementează interfeţele `DataInput` şi `DataOutput`, ceea ce înseamna ca sunt disponibile metode de tipul `read`*XXX*, `write`*XXX*, întocmai ca la clasele `DataInputStream` şi `DataOutputStream`;
- permite atât citirea cât şi scriere din/in fişiere cu acces direct;
- permite specificarea modului de acces al unui fişier (read-only, read-write).

Constructorii acestei clase sunt:

```
RandomAccessFile(StringnumeFisier, StringmodAcces)
  throws IOException
RandomAccessFile(StringnumeFisier, StringmodAcces)
  throws IOException
```

unde *modAcces* poate fi:

- "r" - fişierul este deschis numai pentru citire (read-only)
- "rw" - fişierul este deschis pentru citire şi scriere (read-write)

Exemple:

```
RandomAccesFile f1 = new RandomAccessFile("fisier.txt", "r");
//deschide un fisier pentru citire

RandomAccesFile f2 = new RandomAccessFile("fisier.txt", "rw");
//deschide un fisier pentru scriere si citire
```

Clasa `RandomAccesFile` suportă noţiunea de *pointer de fişier*. Acesta este un indicator ce specifică poziţia curentă în fişier. La deschiderea unui fişier pointerul are valoarea 0, indicând începutul fişierului. Apeluri la metodele de citire/scrirere deplasează pointerul fişierului cu numărul de octeţi citiţi sau scrişi de metodele respective.

In plus faţăde metodele de tip `read` şi `write` clasa pune la dispozitie şi metode pentru controlul poziţiei pointerului de fişier. Acestea sunt:

- `skipBytes` - mută pointerul fişierului înainte cu un număr specificat de octeţi
- `seek` - poziţioneaza pointerului fişierului înaintea unui octet specificat
- `getFilePointer` - returnează poziţia pointerului de fişier.

4.6 Clasa File

Clasa File nu se referă doar la un fişier ci poate reprezenta fie un fişier anume, fie multimea fişierelor dintr-un director.

Specificarea unui fişier/director se face prin specificarea căii absolute spre acel fişier sau a căii relative faţă de directorul curent. Acestea trebuie să respecte convenţiile de specificare a căilor şi numelor fişierelor de pe platforma de lucru.
Utilitate clasei File constă în furnizarea unei modalităţi de a abstractiza dependenţele cailor şi numelor fişierelor faţăde maşina gazdă, precum şi punerea la dispoziţie a unor metode pentru lucrul cu fisere şi directoare la nivelul sistemului de operare.

Astfel, în această clasă vom găsi metode pentru testarea existenţei, ştergerea, redenumirea unui fişier sau director, crearea unui director, listarea fişierelor dintr-un director, etc.

Trebuie menţionat şi faptul că majoritatea constructorilor fluxurilor care permit accesul la fişiere acceptă ca argument un obiect de tip File în locul unui şir ce reprezintă numele fişierului respectiv.

```
File f = new File("fisier.txt");
FileInputStream in = new FileInputStream(f)
```

Cel mai uzual constructor al clasei File este:

```
public File(String numeFisier)
```

Metodele mai importante ale clasei File au denumiri sugestive şi vor fi prezentate prin intermediul exemplului următor care listează fişierele şi subdirectoarele unui director specificat şi, pentru fiecare din ele afişează diverse informaţii:

Listing 4.6: Listarea conţinutului unui director

```
/* Programul listeaza fisierele si subdirectoarele unui
   director.
   Pentru fiecare din ele vor fi afisate diverse informatii.
   Numele directorului este primit ca argument de la
   linia de comanda, sau este directorul curent.
*/

import java.io.*;
```

```
import java.util.*;
public class ListareDirector {

  private static void info(File f) {
    //Afiseaza informatii despre un fisier sau director
    String nume = f.getName();
    if(f.isFile())
      System.out.println("Fisier: " + nume);
    else
    if(f.isDirectory())
      System.out.println("Director: " + nume);

    System.out.println(
      "Cale absoluta: " + f.getAbsolutePath() +
      "\n Poate citi: " + f.canRead() +
      "\n Poate scrie: " + f.canWrite() +
      "\n Parinte: " + f.getParent() +
      "\n Cale: " + f.getPath() +
      "\n Lungime: " + f.length() +
      "\n Data ultimei modificari: " +
        new Date(f.lastModified()));
    System.out.println("--------------");
  }

  public static void main(String[] args) {
    String nume;
    if (args.length == 0)
      nume = "."; //directorul curent
    else
      nume = args[0];

    try {
      File director = new File(nume);
      File[] continut = director.listFiles();

      for(int i = 0; i < continut.length; i++)
        info(continut[i]);

    } catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

Capitolul 5

Interfeţe

5.1 Introducere

5.1.1 Ce este o interfaţă ?

Interfeţele duc conceptul de clasă abstractă cu un pas înainte prin eliminarea oricăror implementări de metode, punând în practică unul din conceptele programării orientate obiect şi anume cel de separare a modelului unui obiect (interfaţă) de implementarea sa. Aşadar, o interfaţă poate fi privita ca un *protocol de comunicare* între obiecte.

O interfaţă Java defineşte un set de metode dar nu specifică nici o implementare pentru ele. O clasă care implementează o interfaţă trebuie obligatoriu să specifice implementări pentru toate metodele interfeţei, supunându-se aşadar unui anumit comportament.

Definiţie

O *interfaţă* este o colecţie de metode fără implementare şi declaraţii de constante.

Interfeţele permit, alături de clase, definirea unor noi tipuri de date.

5.2 Folosirea interfeţelor

5.2.1 Definirea unei interfeţe

Definirea unei interfeţe se face prin intermediul cuvântului cheie **interface**:

```
[public] interface NumeInterfata
    [extends SuperInterfata1, SuperInterfata2...]
{
  /* Corpul interfetei:
     Declaratii de constane
     Declaratii de metode abstracte
  */
}
```

O interfaţă poate avea un singur modificator şi anume `public`. O interfaţă publică este accesibilă tuturor claselor, indiferent de pachetul din care fac parte, implicit nivelul de acces fiind doar la nivelul pachetului din care face parte interfaţa.

O interfaţă poate extinde oricâte interfeţe. Acestea se numesc *superinterfeţe* şi sunt separate prin virgulă. (vezi "Moştenirea multiplă prin intermediul interfeţelor").

Corpul unei interfeţe poate conţine:

- **constante**: acestea pot fi sau nu declarate cu modificatorii `public`, `static` şi `final` care sunt impliciţi, nici un alt modificator neputând apărea în declaraţia unei variabile dintr-o interfaţă. Constantele unei interfeţe trebuie obligatoriu iniţializate.

```
interface Exemplu {
  int MAX = 100;
  // Echivalent cu:
  public static final int MAX = 100;

  int MAX;
  // Incorect, lipseste initializarea

  private int x = 1;
  // Incorect, modificator nepermis
}
```

- **metode fără implementare**: acestea pot fi sau nu declarate cu modificatorul `public`, care este implicit; nici un alt modificator nu poate apărea în declaraţia unei metode a unei interfeţe.

```
interface Exemplu {
  void metoda();
  // Echivalent cu:
  public void metoda();

  protected void metoda2();
  // Incorect, modificator nepermis
```

Atenţie

- Variabilele unei interfeţe sunt implicit publice chiar dacă nu sunt declarate cu modificatorul `public`.
- Variabilele unei interfeţe sunt implicit constante chiar dacă nu sunt declarate cu modificatorii `static` şi `final`.
- Metodele unei interfeţe sunt implicit publice chiar dacă nu sunt declarate cu modificatorul `public`.
- In variantele mai vechi de Java era permis şi modificatorul `abstract` în declaratia interfeţei şi în declaraţiile metodelor, însă acest lucru nu mai este valabil, deoarece atât interfaţa cât şi metodele sale nu pot fi altfel decât abstracte.

5.2.2 Implementarea unei interfeţe

Implementarea uneia sau mai multor interfeţe de către o clasă se face prin intermediul cuvântului cheie **implements**:

```
class NumeClasa implements NumeInterfata
sau
class NumeClasa implements Interfata1, Interfata2, ...
```

O clasă poate implementa oricâte interfeţe sau poate să nu implementeze nici una.

In cazul în care o clasă implementează o anumită interfaţă, atunci trebuie obligatoriu să specifice cod pentru **toate** metodele interfeţei. Din acest motiv, odată creata şi folosită la implementarea unor clase, o interfaţă nu mai trebuie modificată, în sensul că adăugarea unor metode noi sau schimbarea signaturii metodelor existente vor duce la erori în compilarea claselor care o implementează. Evident, o clasă poate avea şi alte metode şi variabile membre în afară de cele definite în interfaţă.

Atenţie

Modificarea unei interfeţe implică modificarea tuturor claselor care implementează acea interfaţă.

O interfaţă nu este o clasă, dar orice referinţă de tip interfaţă poate primi ca valoare o referinţa la un obiect al unei clase ce implementează interfaţa respectivă. Din acest motiv, interfeţele pot fi privite ca tipuri de date şi vom spune adesea că un obiect are tipul X, unde X este o interfaţă, dacă acesta este o instanţă a unei clase ce implementează interfaţa X.

Implementarea unei interfeţe poate să fie şi o clasă abstractă.

5.2.3 Exemplu: implementarea unei stive

Să considerăm următorul exemplu. Dorim să implementăm un nou tip de date numit *Stack*, care să modeleze noţiunea de stivă de obiecte. Obiectele de tip stivă, indiferent de implementarea lor, vor trebui să conţină metodele:

- `push` - adaugă un nou element in stıvă
- `pop` - elimină elementul din vârful stivei
- `peek` - returnează varful stivei
- `empty` - testează dacă stiva este vidă
- `toString` - returnează conţinutul stivei sub forma unui şir de caractere.

Din punctul de vedere al structurii interne, o stivă poate fi implementată folosind un vector sau o listă înlănţuită, ambele soluţii având avantaje şi dezavantaje. Prima soluţie este mai simplu de înţeles, în timp ce a doua este mai eficientă din punctul de vedere al folosirii memoriei. Deoarece nu dorim să legăm tipul de date *Stack* de o anumită implementare structurală, îl vom defini prin intermediul unei interfeţe. Vom vedea imediat avantajele acestei abordări.

Listing 5.1: Interfaţa ce descrie stiva

```
public interface Stack {
   void push(Object item) throws StackException;
   void pop() throws StackException;
   Object peek() throws StackException;
   boolean empty();
   String toString();
}
```

Pentru a trata situaţiile anormale care pot apărea atunci când încercăm să punem un element în stivă şi nu este posibil din lipsă de memorie, sau încercăm să accesăm vârful stivei şi aceasta este vidă, vom defini o excepţie proprie `StackException`:

Listing 5.2: Tipul de excepţie generat de stivă

```
public class StackException extends Exception {
  public StackException() {
    super();
  }
  public StackException(String msg) {
    super(msg);
  }
}
```

Dăm în continuare prima implementare a stivei, folosind un vector:

Listing 5.3: Implementarea stivei folosind un vector

```
// Implementarea stivei folosind un vector de obiecte.
public class StackImpl1 implements Stack {
  private Object items[];
  //Vectorul ce contine obiectele
```

```
  private int n=0;
  //Numarul curent de elemente din stiva

  public StackImpl1(int max) {
    //Constructor
    items = new Object[max];
  }
  public StackImpl1() {
    this(100);
  }
  public void push(Object item) throws StackException {
    if (n == items.length)
      throw new StackException("Stiva este plina!");
    items[n++] = item;
  }
  public void pop() throws StackException {
    if (empty())
      throw new StackException("Stiva este vida!");
    items[--n] = null;
  }
  public Object peek() throws StackException {
    if (empty())
      throw new StackException("Stiva este vida!");
    return items[n-1];
  }
  public boolean empty() {
    return (n==0);
  }
  public String toString() {
    String s="";
    for(int i=n-1; i>=0; i--)
      s += items[i].toString() + " ";
    return s;
  }
}
```

Remarcaţi că, deşi în interfaţă metodele nu sunt declarate explicit cu modificatorul `public`, ele sunt totuşi publice şi trebuie declarate ca atare în clasă.
Trebuie remarcat şi faptul că metoda `toString` este definită deja în clasa `Object`, deci clasa noastră o are deja implementată şi nu am fi obţinut nici o eroare la compilare dacă nu o implementam explicit. Ceea ce facem acum este de fapt supradefinirea ei.

O altă observaţie importantă se referă la faptul că trebuie să declarăm în cadrul interfeţei şi excepţiile aruncate de metode, ce trebuie obligatoriu tratate.

Să vedem acum modalitatea de implementare a stivei folosind o listă înlănţuită:

Listing 5.4: Implementarea stivei folosind o listă

```
// Implementarea stivei folosind o lista inlantuita.
public class StackImpl2 implements Stack {

  class Node {
    //Clasa interna ce reprezinta un nod al listei
    Object item; //informatia din nod
    Node link;   //legatura la urmatorul nod
    Node(Object item, Node link) {
      this.item = item;
      this.link = link;
    }
  }

  private Node top=null;
  //Referinta la varful stivei

  public void push(Object item) {
    Node node = new Node(item, top);
    top = node;
  }
   public void pop() throws StackException {
     if (empty())
       throw new StackException("Stiva este vida!");
     top = top.link;
   }
   public Object peek() throws StackException {
     if (empty())
       throw new StackException("Stiva este vida!");
     return top.item;
   }
   public boolean empty() {
     return (top == null);
   }
   public String toString() {
     String s="";
     Node node = top;
     while (node != null) {
```

```
        s += (node.item).toString() + " ";
        node = node.link;
      }
      return s;
    }
}
```

Singura observaţie pe care o facem aici este că, deşi metoda `push` din interfaţă declară aruncarea unor excepţii de tipul `StackException`, nu este obligatoriu ca metoda din clasă să specifice şi ea acest lucru, atât timp cât nu generează excepţii de acel tip. Invers este însă obligatoriu.

In continuare este prezentată o mică aplicaţie demonstrativă care foloseşte tipul de date nou creat şi cele două implementări ale sale:

Listing 5.5: Folosirea stivei

```
public class TestStiva {

  public static void afiseaza(Stack s) {
    System.out.println("Continutul stivei este: " + s);
  }

  public static void main(String args[]){
    try {
      Stack s1 = new StackImpl1();
      s1.push("a");
      s1.push("b");
      afiseaza(s1);

      Stack s2 = new StackImpl2();
      s2.push(new Integer(1));
      s2.push(new Double(3.14));
      afiseaza(s2);


    } catch (StackException e) {
        System.err.println("Eroare la lucrul cu stiva!");
        e.printStackTrace();
    }
  }
}
```

Observaţi folosirea interfeţei `Stack` ca un tip de date, ce aduce flexibilitate sporită în manevrarea claselor ce implementează tipul respectiv. Metoda

`afiseaza` acceptă ca argument orice obiect al unei clase ce implementează `Stack`.

Observaţie
In pachetul `java.util` există clasa `Stack` care modelează noţiune de stivă de obiecte şi, evident, aceasta va fi folosită în aplicaţiile ce au nevoie de acest tip de date. Exemplu oferit de noi nu are legătură cu această clasă şi are rol pur demonstrativ.

5.3 Interfeţe şi clase abstracte

La prima vedere o interfaţă nu este altceva decât o clasă abstractă în care toate metodele sunt abstracte (nu au nici o implementare).
Aşadar, *o clasă abstractă nu ar putea înlocui o interfaţă* ?
Raspunsul la intrebare depinde de situaţie, însă în general este *'Nu'*.
Deosebirea constă în faptul că unele clase sunt forţate să extindă o anumită clasă (de exemplu orice applet trebuie să fie subclasa a clasei `Applet`) şi nu ar mai putea sa extindă o altă clasă, deoarece în Java nu exista decât moştenire simpla. Fara folosirea interfeţelor nu am putea forţa clasa respectivă să respecte diverse tipuri de protocoale.

La nivel conceptual, diferenţa constă în:

- extinderea unei clase abstracte forţează o relaţie între clase;
- implementarea unei interfeţe specifică doar necesitatea implementării unor anumie metode.

In multe situaţii interfeţele şi clasele abstracte sunt folosite împreună pentru a implementa cât mai flexibil şi eficient o anumită ierarhie de clase. Un exemplu sugestiv este dat de clasele ce descriu colecţii. Ca sa particularizăm, există:

- interfaţa **List** care impune protocolul pe care trebuie să îl respecte o clasă de tip listă,
- clasa abstractă **AbstractList** care implementează interfaţa `List` şi oferă implementări concrete pentru metodele comune tuturor tipurilor de listă,

- clase concrete, cum ar fi **LinkedList**, **ArrayList** care extind `AbstractList`.

5.4 Moştenire multiplă prin interfeţe

Interfeţele nu au nici o implementare şi nu pot fi instanţiate. Din acest motiv, nu reprezintă nici o problemă ca anumite clase să implementeze mai multe interfeţe sau ca o interfaţă să extindă mai multe interfeţe (să aibă mai multe superinterfeţe)

```
class NumeClasa implements Interfata1, Interfata2, ...
interface NumeInterfata extends Interfata1, Interfata2, ...
```

O interfaţă mosteneste atât constantele cât şi declaraţiile de metode de la superinterfeţele sale. O clasă moşteneste doar constantele unei interfeţe şi responsabilitatea implementării metodelor sale.

Să considerăm un exemplu de clasa care implementeaza mai multe interfeţe:

```
interface Inotator {
  void inoata();
}
interface Zburator {
  void zboara();
}
interface Luptator {
  void lupta();
}
class Erou implements Inotator, Zburator, Luptator {
  public void inoata() {}
  public void zboara() {}
  public void lupta()  {}
}
```

Exemplu de interfaţă care extinde mai multe interfeţe:

```
interface Monstru {
  void ameninta();
}
interface MonstruPericulos extends Monstru {
  void distruge();
```

```
  }
  interface Mortal {
    void omoara();
  }
  interface Vampir extends MonstruPericulos, Mortal {
    void beaSange();
  }
  class Dracula implements Vampir {
    public void ameninta() {}
    public void distruge() {}
    public void omoara()() {}
    public void beaSange() {}
  }
```

Evident, pot apărea situaţii de ambiguitate, atunci când există constante sau metode cu aceleaşi nume în mai multe interfeţe, însă acest lucru trebuie întotdeauna evitat, deoarece scrierea unui cod care poate fi confuz este un stil prost de programare. In cazul in care acest lucru se întâmplă, compilatorul nu va furniza eroare decât dacă se încearcă referirea constantelor ambigue fără a le prefixa cu numele interfeţei sau dacă metodele cu acelaşi nume nu pot fi deosbite, cum ar fi situaţia când au aceeaşi listă de argumente dar tipuri returnate incompatibile.

```
interface I1 {
  int x=1;
  void metoda();
}

interface I2 {
  int x=2;
  void metoda();   //corect
  //int metoda();  //incorect
}

class C implements I1, I2 {
  public void metoda() {
    System.out.println(I1.x); //corect
    System.out.println(I2.x); //corect
    System.out.println(x);    //ambiguitate
```

```
  }
}
```

Să recapitulăm câteva lucruri legate de clase şi interfeţe:

- O clasă nu poate avea decât o superclasă.
- O clasă poate implementa oricâte interfeţe.
- O clasă trebuie obligatoriu să trateze metodele din interfeţele pe care la implementează.
- Ierarhia interfeţelor este independentă de ierarhia claselor care le implementează.

5.5 Utilitatea interfeţelor

După cum am văzut, o interfaţă defineşte un protocol ce poate fi implementat de orice clasă, indiferent de ierarhia de clase din care face parte. Interfeţele sunt utile pentru:

- definirea unor similaritati între clase independente fără a forţa artificial o legatură între ele;
- asigură că toate clasele care implementează o interfaţă pun la dipoziţie metodele specificate în interfaţă - de aici rezultă posibilitatea implementării unor clase prin mai multe modalităţi şi folosirea lor într-o manieră unitară;
- definirea unor grupuri de constante;
- transmiterea metodelor ca parametri;

5.5.1 Crearea grupurilor de constante

Deoarece orice variabilă a unei interfeţe este implicit declarată cu `public`, `static` si `final`, interfeţele reprezintă o metodă convenabilă de creare a unor grupuri de constante care să fie folosite global într-o aplicaţie:

```
public interface Luni {
  int IAN=1, FEB=2, ..., DEC=12;
}
```

Folosirea acestor constante se face prin expresii de genul `NumeInterfata.constanta`, ca în exemplul de mai jos:

```
if (luna < Luni.DEC)
  luna ++
else
  luna = Luni.IAN;
```

5.5.2 Transmiterea metodelor ca parametri

Deoarece nu există pointeri propriu-zişi, transmiterea metodelor ca parametri este realizată în Java prin intermediul interfeţelor. Atunci când o metodă trebuie să primească ca argument de intrare o referinţă la o altă funcţie necesară execuţiei sale, cunoscută doar la momentul execuţiei, atunci argumentul respectiv va fi declarat de tipul unei interfeţe care conţine metoda respectivă. La execuţie metoda va putea primi ca parametru orice obiect ce implementează interfaţa respectivă şi deci conţine şi codul funcţiei respective, aceasta urmând să fie apelată normal pentru a obţine rezultatul dorit.

Această tehnică, denumită şi *call-back*, este extrem de folosită în Java şi trebuie neapărat înţeleasă. Să considerăm mai multe exemple pentru a clarifica lucrurile.

Primul exemplu se referă la explorarea nodurilor unui graf. In fiecare nod trebuie să se execute prelucrarea informaţiei din nodul respectiv prin intermediul unei funcţii primite ca parametru. Pentru aceasta, vom defini o interfaţă `Functie` care va specifica metoda trimisă ca parametru.

```
interface Functie {
  public void executa(Nod u);
}

class Graf {
  //...
  void explorare(Functie f) {
    //...
```

```
    if (explorarea a ajuns in nodul v) {
      f.executa(v);
      //...
    }
  }
}

//Definim doua functii
class AfisareRo implements Functie {
  public void executa(Nod v) {
    System.out.println("Nodul curent este: " + v);
  }
}
class AfisareEn implements Functie {
  public void executa(Nod v) {
    System.out.println("Current node is: " + v);
  }
}

public class TestCallBack {
  public static void main(String args[]) {
    Graf G = new Graf();
    G.explorare(new AfisareRo());
    G.explorare(new AfisareEn());
  }
}
```

Al doilea xemplu va fi prezentat în secţiunea următoare, întrucât face parte din API-ul standard Java şi vor fi puse în evidenţă, prin intermediul său, şi alte tehnici de programare.

5.6 Interfaţa `FilenameFilter`

Instanţele claselor ce implementează aceasta interfaţă sunt folosite pentru a crea filtre pentru fişiere şi sunt primite ca argumente de metode care listează conţinutul unui director, cum ar fi metoda `list` a clasei `File`.
Aşadar, putem spune că metoda `list` primeşte ca argument o altă funcţie care specifică dacă un fişier va fi returnat sau nu (criteriul de filtrare).

Interfaţa `FilenameFilter` are o singură metodă: **accept** care specifică criteriul de filtrare şi anume, testează dacă numele fişierului primit ca parametru îndeplineşte condiţiile dorite de noi.

Definiţia interfeţei este:

```
public interface FilenameFilter {
  public boolean accept(File dir, String numeFisier);
}
```

Aşadar, orice clasă de specificare a unui filtru care implementeză interfaţa `FilenameFilter` trebuie să implementeze metoda `accept` a acestei interfeţe. Aceste clase mai pot avea şi alte metode, de exemplu un constructor care să primeasca criteriul de filtrare. In general, o clasă de specificare a unui filtru are următorul format:

```
class FiltruFisiere implements FilenameFilter {
  String filtru;

  // Constructorul
  FiltruFisiere(String filtru) {
    this.filtru = filtru;
  }

  // Implementarea metodei accept
  public boolean accept(File dir, String nume) {
    if (filtrul este indeplinit)
      return true;
    else
      return false;
  }
}
```

Metodele cele mai uzuale ale clasei `String` folosite pentru filtrarea fişierelor sunt:

- **endsWith** - testează dacă un şir are o anumită terminaţie
- **indexOf** - testează dacă un şir conţine un anumit subşir, returnând poziţia acestuia, sau 0 în caz contrar.

Instanţele claselor pentru filtrare sunt primite ca argumente de metode de listare a conţinutului unui director. O astfel de metodă este `list` din clasa `File`:

```
String[] list (FilenameFilter filtru)
```

Observati că aici interfaţa este folosită ca un tip de date, ea fiind substituită cu orice clasă care o implementează. Acesta este un exemplu tipic de transmitere a unei funcţii (funcţia de filtrare `accept`) ca argument al unei metode.

Să considerăm exemplul complet în care dorim să listăm fişierele din directorul curent care au o anumită extensie.

Listing 5.6: Listarea fişierelor cu o anumită extensie

```
/* Listarea fisierelor din directorul curent
   care au anumita extensie primita ca argument.
   Daca nu se primeste nici un argument, vor fi listate toate
      .
*/
import java.io.*;
class Listare {

  public static void main(String[] args) {
    try {
      File director = new File(".");
      String[] list;
      if (args.length > 0)
        list = director.list(new Filtru(args[0]));
      else
        list = director.list();

      for(int i = 0; i < list.length; i++)
        System.out.println(list[i]);
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
}

class Filtru implements FilenameFilter {
  String extensie;
  Filtru (String extensie) {
    this.extensie = extensie;
```

```
  }
  public boolean accept (File dir, String nume) {
    return ( nume.endsWith("." + extensie) );
  }
}
```

5.6.1 Folosirea claselor anonime

In cazul în care nu avem nevoie de clasa care reprezintă filtrul pentru listarea fişierelor dintr-un director decât o singură dată, pentru a evita crearea unei noi clase de sine stătătoare care să fie folosită pentru instanţierea unui singur obiect, putem folosi clasă internă anonimă, această situaţie fiind un exemplu tipic de folosire a acestora.

Listing 5.7: Folosirea unei clase anonime

```
/* Listarea fisierelor din directorul curent
   folosind o clasa anonima pentru filtru.
*/
import java.io.*;
class Listare {

  public static void main(String[] args) {
    try {
      File director = new File(".");
      String[] list;
      if (args.length > 0) {
        final String extensie = args[0];
        list = director.list(new FilenameFilter() {
          // Clasa interna anonima
          public boolean accept (File dir, String nume) {
            return ( nume.endsWith("." + extensie) );
          }
        });
      }
      else
        list = director.list();

      for(int i = 0; i < list.length; i++)
        System.out.println(list[i]);
    } catch(Exception e) {
      e.printStackTrace();
    }
```

```
  }
}
```

Aşadar, o modalitate uzuală de folosire a claselor anonime pentru instanţierea unui obiect care trebuie să respecte o interfaţă este:

```
metoda(new Interfata() {
  // Implementarea metodelor interfetei
});
```

5.7 Compararea obiectelor

Am văzut în primul capitol că o soluţie facilă şi eficientă de sortare a unui vector este folosirea metodei `sort` din clasa `java.util.Arrays`.

```
int v[]={3, 1, 4, 2};
java.util.Arrays.sort(v);
// Sorteaza vectorul v
// Acesta va deveni {1, 2, 3, 4}
```

In cazul în care elementele din vector sunt de tip primitiv, ca in exemplul de mai sus, nu există nici o problemă în a determina ordinea firească a elementelor. Ce se întâmplă însă atunci când vectorul conţine referinţe la obiecte de un anumit tip ? Să considerăm următorul exemplu, în care dorim să sortăm un vector format din instanţe ale clasei `Persoana`, definită mai jos:

Listing 5.8: Clasa Persoana (fără suport pentru comparare)

```
class Persoana {
  int cod;
  String nume;

  public Persoana(int cod, String nume) {
    this.cod = cod;
    this.nume = nume;
  }

  public String toString() {
     return cod + " \t " + nume;
  }
}
```

Programul următor ar trebui să sorteze un vector de persoane:

Listing 5.9: Sortarea unui vector de tip referinţă

```
class Sortare {
  public static void main(String args[]) {
    Persoana p[] = new Persoana[4];
    p[0] = new Persoana(3, "Ionescu");
    p[1] = new Persoana(1, "Vasilescu");
    p[2] = new Persoana(2, "Georgescu");
    p[3] = new Persoana(4, "Popescu");

    java.util.Arrays.sort(p);

    System.out.println("Persoanele ordonate dupa cod:");
    for(int i=0; i<p.length; i++)
      System.out.println(p[i]);
  }
}
```

La execuţia acestei aplicaţii va fi obţinută o excepţie, deoarece metoda `sort` nu ştie care este ordinea naturală a obiectelor de tip `Persoana`. Va trebui, într-un fel sau altul, să specificăm acest lucru.

5.7.1 Interfaţa `Comparable`

Interfaţa `Comparable` impune o ordine totală asupra obiectelor unei clase ce o implementează. Această ordine se numeşte *ordinea naturală* a clasei şi este specificată prin intermediul metodei **compareTo**. Definiţia interfeţei este:

```
public interface Comparable {
  int compareTo(Object o);
}
```

Aşadar, o clasă ale cărei instanţe trebuie să fie comparabil va implementa metoda `compareTo` care trebuie să returneze:

- **o valoare strict negativă**: dacă obiectul curent (`this`) este mai *mic* decâ obiectul primit ca argument;
- **zero**: dacă obiectul curent este *egal* decâ obiectul primit ca argument;

- **o valoare strict pozitivă**: dacă obiectul curent este mai *mare* decâ obiectul primit ca argument.

Reamintim că metoda `equals`, moştenită din `Object` de toate clasele, determină dacă două obiecte sunt egale (au aceeaşi valoare). Spunem că ordinea naturală a unei clase C este *consitentă* cu `equals` dacă şi numai dacă `(e1.compareTo((Object)e2) == 0)` are aceeasşi valoare logică cu `e1.equals((Object)e2`, pentru orice $e1$, $e2$ instanţe ale lui C.

`null` nu este instanţă a nici unei clase şi `e.compareTo(null)` trebuie să arunce o excepţie de tip `NullPointerException` chiar dacă `e.equals(null)` returnează `false`.

Să presupunem că dorim ca ordinea naturală a persoanelor să fie după codul lor intern.

Listing 5.10: Clasa Persoana cu suport pentru comparare

```
class Persoana implements Comparable {
  int cod;
  String nume;

  public Persoana(int cod, String nume) {
    this.cod = cod;
    this.nume = nume;
  }

  public String toString() {
     return cod + " \t " + nume;
  }

  public boolean equals(Object o) {
    if (!(o instanceof Persoana))
      return false;
    Persoana p = (Persoana) o;
    return (cod == p.cod) && (nume.equals(p.nume));
  }

  public int compareTo(Object o) {
    if (o==null)
      throw new NullPointerException();
    if (!(o instanceof Persoana))
      throw new ClassCastException("Nu pot compara!");

    Persoana p = (Persoana) o;
```

```
    return (cod - p.cod);
  }
}
```

Observaţi folosirea operatorului `instanceof`, care verifică dacă un obiect este instanţă a unei anumite clase. Metoda `compareTo` va arunca o excepţie de tipul `ClassCastException` dacă se încearcă compararea unui obiect de tip `Persoana` cu un obiect de alt tip. Metoda `equals` va returna, pur şi simplu, `false`.

5.7.2 Interfaţa `Comparator`

In cazul în care dorim să sortăm elementele unui vector ce conţine referinţe după alt criteriu decât ordinea naturală a elemenetelor, avem nevoie de o altă soluţie. Aceasta este oferită tot de metoda `sort` din clasa `java.util.Arrays`, dar în varianta în care, pe lângă vectorul ce trebuie sortat, vom transmite un argument de tip `Comparator` care să specifice modalitatea de comparare a elementelor.

Interfaţa `java.util.Comparator` conţine metoda **compare**, care impune o ordine totală asupra elementelor unei colecţii. Aceasta returnează un întreg cu aceeaşi semnificaţie ca la metoda `compareTo` a interfeţei `Comparator` şi are următoarea definiţie: `int compare(Object o1, Object o2);`

Să presupunem că dorim să sortăm persoanele ordonate după numele lor. Pentru definirea comparatorului vom folosi o clasă anonimă.

Listing 5.11: Sortarea unui vector folosind un comparator

```
import java.util.*;
class Sortare {
  public static void main(String args[]) {
    Persoana p[] = new Persoana[4];
    p[0] = new Persoana(3, "Ionescu");
    p[1] = new Persoana(1, "Vasilescu");
    p[2] = new Persoana(2, "Georgescu");
    p[3] = new Persoana(4, "Popescu");

    Arrays.sort(p, new Comparator() {
      public int compare(Object o1, Object o2) {
        Persoana p1 = (Persoana)o1;
        Persoana p2 = (Persoana)o2;
        return (p1.nume.compareTo(p2.nume));
```

```
      }
    });
    System.out.println("Persoanele ordonate dupa nume:");
    for(int i=0; i<p.length; i++)
      System.out.println(p[i]);
  }
}
```

Observaţi cum compararea a două şiruri de caractere se face tot cu metoda `compareTo`, clasa `String` implemenând interfaţa `Comparable`.

5.8 Adaptori

In cazul în care o interfaţă conţine mai multe metode şi, la un moment dat, avem nevoie de un obiect care implementează interfaţa respectiv dar nu specifică cod decât pentru o singură metodă, el trebui totuşi să implementeze toate metodele interfeţei, chiar dacă nu specifică nici un cod.

```
interface X {
  void metoda_1();
  void metoda_2();
  ...
  void metoda_n();
}


...
// Avem nevoie de un obiect de tip X
// ca argument al unei functii
functie(new X() {
  public void metoda_1() {
    // Singura metoda care ne intereseaza
    ...
  }
  // Trebuie sa apara si celelalte metode
  // chiar daca nu au implementare efectiva
  public void metoda_2() {}
  public void metoda_3() {}
  ...
  public void metoda_n() {}
```

```
});
```

Această abordare poate fi neplăcută dacă avem frecvent nevoie de obiecte ale unor clase ce implementează interfaţa X. Soluţia la această problemă este folosirea **adaptorilor**.

Definiţie
Un *adaptor* este o clasă abstractă care implementează o anumită interfaţă fără a specifica cod nici unei metode a interfeţei.

```
public abstract class XAdapter implements X {
  public void metoda_1() {}
  public void metoda_2() {}
  ...
  public void metoda_n() {}
}
```

In situaţia când avem nevoie de un obiect de tip X vom folosi clasa abstractă, supradefinind doar metoda care ne interesează:

```
functie(new XAdapter() {
  public void metoda_1() {
    // Singura metoda care ne intereseaza
    ...
  }
});
```

Mai multe exemple de folosire a adaptorilor vor fi date în capitolul "Interfaţa grafică cu utilizatorul".

Capitolul 6

Organizarea claselor

6.1 Pachete

Definiţie

Un *pachet* este o colecţie de clase şi interfeţe înrudite din punctul de vedere al funcţionalităţii lor. Sunt folosite pentru găsirea şi utilizarea mai uşoară a claselor, pentru a evita conflictele de nume şi pentru a controla accesul la anumite clase. In alte limbaje de programare pachetele se mai numesc librării sau bibilioteci.

6.1.1 Pachetele standard (J2SDK)

Platforma standard de lucru Java se bazează pe o serie de pachete cu ajutorul cărora se pot construi într-o manieră simplificată aplicaţiile. Există deci un set de clase deja implementate care modelează structuri de date, algoritmi sau diverse noţiuni esenţiale în dezvoltarea unui program. Cele mai importante pachete şi suportul oferit de lor sunt:

- **java.lang** - clasele de bază ale limbajului Java
- **java.io** - intrări/ieşiri, lucrul cu fişiere
- **java.util** - clase şi interfeţe utile
- **java.applet** - dezvoltarea de appleturi

- **java.awt** - interfaţa grafică cu utilizatorul
- **java.awt.event** - mecanismele de tratare e evenimentelor generate de utilizator
- **java.beans** - scrierea de componente reutilizabile
- **java.net** - programare de reţea
- **java.sql** - lucrul cu baze de date
- **java.rmi** - execuţie la distanţă *Remote Message Interface*
- **java.security** - mecanisme de securitate: criptare, autentificare
- **java.math** - operaţii matematice cu numere mari
- **java.text** - lucrul cu texte, date şi numere independent de limbă
- **java.lang.reflect** - introspecţie
- **javax.swing** - interfaţa grafică cu utilizatorul, mult îmbogăţită faţă de AWT.
- **...**

6.1.2 Folosirea membrilor unui pachet

Conform specificaţiilor de acces ale unei clase şi ale mebrilor ei, doar clasele publice şi membrii declaraţi publici ai unei clase sunt accesibili în afara pachetului în care se găsesc. După cum am văzut în secţiunea "Specificatori de acces pentru membrii unei clase", accesul implicit în Java este la nivel de *pachet*.

Pentru a folosi o clasă publică dintr-un anumit pachet, sau pentru a apela o metodă publică a unei clase publice a unui pachet, există trei soluţii:

- specificarea numelui complet al clasei
- importul clasei respective
- importul întregului pachet în care se găseşte clasa.

Specificarea *numelui complet* al clasei se face prin prefixarea *numelui scurt* al clasei cu numele pachetului din care face parte: `numePachet.NumeClasa`.

```
Button           - numele scurt al clasei
java.awt         - pachetul din care face parte
java.awt.Button  - numele complet al clasei
```

Această metodă este recomandată doar pentru cazul în care folosirea acelei clase se face o singură dată sau foarte rar.

De exemplu, ar fi extrem de neplăcut să scriem de fiecare dată când vrem să declarăm un obiect grafic secvenţe de genul:

```
java.awt.Button b1 = new  java.awt.Button("OK");
java.awt.Button b2 = new  java.awt.Button("Cancel");
java.awt.TextField tf1 = new java.awt.TextField("Neplacut");
java.awt.TextField tf2 = new java.awt.TextField("Tot neplacut");
```

In aceste situaţii, vom importa în aplicaţia noastră clasa respectivă, sau întreg pachetul din care face parte. Acest lucru se realizează prin instrucţiunea **import**, care trebuie să apară la începutul fişierelor sursă, înainte de declararea vreunei clase sau interfeţe.

6.1.3 Importul unei clase sau interfeţe

Se face prin instrucţiunea **import** în care specificăm numele complet al clasei sau interfeţei pe care dorim să o folosim dintr-un anumit pacehet:

```
import numePachet.numeClasa;

//Pentru exemplul nostru:
import java.awt.Button;
import java.awt.TextField;
```

Din acest moment, vom putea folosi în clasele fişierului în care am plasat instrucţiunea de import numele scurt al claselor `Button` şi `TextField`:

```
Button b1 = new  Button("OK");
Button b2 = new  Button("Cancel");
TextField tf1 = new TextField("Placut");
TextField tf2 = new TextField("Foarte placut");
```

Această abordare este eficientă şi recomandată în cazul în care nu avem nevoie decât de câteva clase din pachetul respectiv. Dacă în exemplul nostru am avea nevoie şi de clasele `Line`, `Point`, `Rectangle`, `Polygon`, ar trebui să avem câte o instrucţiune de import pentru fiecare dintre ele:

```
import java.awt.Button;
import java.awt.TextField;
import java.awt.Rectangle;
import java.awt.Line;
import java.awt.Point;
import java.awt.Polygon;
```

In această situaţie ar fi mai simplu să folosim importul la cerere din întregul pachet şi nu al fiecărei clase în parte.

6.1.4 Importul la cerere dintr-un pachet

Importul la cerere dintr-un anumit pachet se face printr-o instrucţiune `import` în care specificăm numele pachetului ale cărui clase şi interfeţe dorim să le folosim, urmat de simbolul `*`. Se numeşte *import la cerere* deoarece încărcarea claselor se face dinamic, în momentul apelării lor.

```
import numePachet.*;

//Pentru exemplul nostru:
import java.awt.*;
```

Din acest moment, vom putea folosi în clasele fişierului în care am plasat instrucţiunea de import numele scurt al tuturor claselor pachetului importat:

```
Button b = new Button("OK");
Point p = new Point(0, 0);
```

Atenţie

* nu are semnificaţia uzuală de la fişiere de wildcard (mască) şi nu poate fi folosit decât ca atare. O expresie de genul `import java.awt.C*;` va produce o eroare de compilare.

In cazul în care sunt importate două sau mai multe pachete care conţin clase (interfeţe) cu acelaşi nume, atunci referirea la ele trebuie făcută doar folosind numele complet, în caz contrar fiind semnalată o *ambiguitate* de către compilator.

```
import java.awt.*;
// Contine clasa List

import java.util.*;
// Contine interfata List

...
List x;   //Declaratie ambigua

java.awt.List  a = new java.awt.List(); //corect
java.util.List b = new ArrayList();     //corect
```

Sunt considerate importate automat, pentru orice fişier sursă, următoarele pachete:

- pachetul `java.lang`

  ```
  import java.lang.*;
  // Poate sau nu sa apara
  // Mai bine nu...
  ```

- pachetul curent
- pachetul implicit (fără nume)

6.1.5 Importul static

Această facilitate, introdusă începând cu versiunea 1.5, permite referirea constantelor statice ale unei clase fără a mai specifica numele complet al acesteia şi este implementată prin adăugarea cuvântului cheie **static** după cel de `import`:

```
import static numePachet.NumeClasa.*;
```

Astfel, în loc să ne referim la constantele clasei cu expresii de tipul `NumeClasa.CONSTANTA`, putem folosi doar numele constantei.

```
// Inainte de versiuna 1.5
import java.awt.BorderLayout.*;
...
fereastra.add(new Button(), BorderLayout.CENTER);

// Incepand cu versiunea 1.5
import java.awt.BorderLayout.*;
import static java.awt.BorderLayout.*;
...
fereastra.add(new Button(), CENTER);
```

Atenţie

Importul static nu importă decât constantele statice ale unei clase, nu şi clasa în sine.

6.1.6 Crearea unui pachet

Toate clasele şi interfeţele Java apartin la diverse pachete, grupate după funcţionalitatea lor. După cum am văzut clasele de bază se găsesc în pachetul `java.lang`, clasele pentru intrări/ieşiri sunt în `java.io`, clasele pentru interfaţa grafică în `java.awt`, etc.

Crearea unui pachet se realizează prin scriere la începutul fişierelor sursă ce conţin clasele şi interfeţele pe care dorim să le grupăm într-un pachet a instrucţiunii: **package numePachet;**

Să considerăm un exemplu: presupunem că avem două fişiere sursă `Graf.java` şi `Arbore.java`.

```
//Fisierul Graf.java
  package grafuri;
  class Graf {...}
  class GrafPerfect extends Graf {...}

//Fisierul Arbore.java
  package grafuri;
  class Arbore {...}
```

```
class ArboreBinar extends Arbore {...}
```

Clasele `Graf`, `GrafPerfect`, `Arbore`, `ArboreBinar` vor face parte din acelaşi pachet `grafuri`.

Instrucţiunea `package` acţionează asupra întregului fişier sursă la începutul căruia apare. Cu alte cuvinte nu putem specifica faptul că anumite clase dintr-un fişier sursă aparţin unui pachet, iar altele altui pachet.

Dacă nu este specificat un anumit pachet, clasele unui fişier sursă vor face parte din pachetul implicit (care nu are nici un nume). In general, pachetul implicit este format din toate clasele şi intefeţele directorului curent de lucru. Este recomandat însă ca toate clasele şi intefetele să fie plasate în pachete, pachetul implicit fiind folosit doar pentru aplicaţii mici sau prototipuri.

6.1.7 Denumirea unui pachet

Există posibilitatea ca doi programatori care lucrează la un proiect comun să folosească acelaşi nume pentru unele din clasele lor. De asemenea, se poate ca una din clasele unei aplicaţii să aibă acelaşi nume cu o clasă a mediului Java. Acest lucru este posibil atât timp cât clasele cu acelaşi nume se gasesc în pachete diferite, ele fiind diferenţiate prin prefixarea lor cu numele pachetelor.

Ce se întâmplă însă când doi programatori care lucrează la un proiect comun folosesc clase cu acelaşi nume, ce se gasesc în pachete cu acelaşi nume ?

Pentru a evita acest lucru, companiile folosesc inversul domeniului lor Internet în denumirea pachetelor implementate în cadrul companiei, cum ar fi `ro.companie.numePachet`. In cadrul aceleiasi companii, conflictele de nume vor fi rezolvate prin diverse convenţii de uz intern, cum ar fi folosirea numelui de cont al programatorilor în denumirea pachetelor create de aceştia. De exemplu, programatorul cu numele Ion al companiei XSoft, având contul ion@xsoft.ro, îşi va prefixa pachetele cu `ro.xsoft.ion`, pentru a permite identificarea în mod unic a claselor sale, indiferent de contextul în care acestea vor fi integrate.

6.2 Organizarea fişierelor

6.2.1 Organizarea fişierelor sursă

Orice aplicaţie nebanală trebuie să fie construită folosind o organizare ierarhică a componentelor sale. Este recomandat ca strategia de organizare a fişierelor sursă să respecte următoarele convenţii:

- Codul sursă al claselor şi interfeţelor să se gasească în fişiere ale căror nume să fie chiar numele lor scurt şi care să aibă extensia `.java`.

Atenţie

Este **obligatoriu** ca o clasă/interfaţă publică să se gasească într-un fişier având numele clasei(interfeţei) şi extenisa `.java`, sau compilatorul va furniza o eroare. Din acest motiv, într-un fişier sursă nu pot exista două clase sau interfeţe publice. Pentru clasele care nu sunt publice acest lucru nu este obligatoriu, ci doar recomandat. Intr-un fişier sursă pot exista oricâte clase sau interfeţe care nu sunt publice.

- Fişierele sursă trebuie să se găsească în directoare care să reflecte numele pachetelor în care se găsesc clasele şi interfeţele din acele fişiere. Cu alte cuvinte, un director va conţine surse pentru clase şi interfeţe din acelaşi pachet iar numele directorului va fi chiar numele pachetului. Dacă numele pachetelor sunt formate din mai multe unităţi lexicale separate prin punct, atunci acestea trebuie de asemenea să corespundă unor directoare ce vor descrie calea spre fişierele sursă ale căror clase şi interfeţe fac parte din pachetele respective.

Vom clarifica modalitatea de organizare a fişierelor sursă ale unei aplicatii printr-un exemplu concret. Să presupunem că dorim crearea unor componente care să reprezinte diverse noţiuni matematice din domenii diferite, cum ar fi geometrie, algebră, analiză, etc. Pentru a simplifica lucrurile, să presupunem că dorim să creăm clase care să descrie următoarele notiuni: *poligon, cerc, poliedru, sferă, grup, funcţie.*
O primă variantă ar fi să construim câte o clasă pentru fiecare şi să le plasăm

în acelaşi director împreuna cu un program care să le folosească, însă, având în vedere posibila extindere a aplicaţiei cu noi reprezentări de noţiuni matematice, această abordare ar fi ineficientă.

O abordare elegantă ar fi aceea în care clasele care descriu noţiuni din acelaşi domeniu sa se gasească în pachete separate şi directoare separate. Ierarhia fişierelor sursa ar fi:

```
/matematica
  /surse
    /geometrie
      /plan
        Poligon.java
        Cerc.java
      /spatiu
        Poliedru.java
        Sfera.java
    /algebra
      Grup.java
    /analiza
      Functie.java

    Matematica.java
```

Clasele descrise în fişierele de mai sus trebuie declarate în pachete denumite corespunzator cu numele directoarelor în care se gasesc:

```
// Poligon.java
package geometrie.plan;
public class Poligon { . . . }

// Cerc.java
package geometrie.plan;
public class Cerc { . . . }


// Poliedru.java
package geometrie.spatiu;
public class Poliedru { . . . }
```

```
// Sfera.java
package geometrie.spatiu;
public class Sfera { . . . }

// Grup.java
package algebra;
public class Grup { . . . }

// Functie.java
package analiza;
public class Functie { . . . }
```

`Matematica.java` este clasa principală a aplicaţiei.
După cum se observă, numele lung al unei clase trebuie să descrie calea spre acea clasă în cadrul fişierelor sursă, relativ la directorul în care se găseşte aplicaţia.

6.2.2 Organizarea unităţilor de compilare (`.class`)

In urma compilării fişierelor sursă vor fi generate unităţi de compilare pentru fiecare clasă şi interfaţă din fişierele sursă. După cum ştim acestea au extensia `.class` şi numele scurt al clasei sau interfeţei respective.

Spre deosebire de organizarea surselor, un fişier `.class` **trebuie** să se gaseasca într-o ierarhie de directoare care să reflecte numele pachetului din care face parte clasa respectivă.

Implicit, în urma compilării fişierele sursă şi unităţile de compilare se găsesc în acelaşi director, însă ele pot fi apoi organizate separat. Este recomandatînsă ca această separare să fie făcută automat la compilare.

Revenind la exemplul de mai sus, vom avea următoarea organizare:

```
/matematica
  /clase
    /geometrie
      /plan
        Poligon.class
        Cerc.class
      /spatiu
```

```
        Poliedru.class
        Sfera.class
    /algebra
      Grup.class
    /analiza
      Functie.class

    Matematica.class
```

Crearea acestei structuri ierarhice este facută automat de către compilator. In directorul aplicatiei (`matematica`) creăm subdirectorul `clase` şi dăm comanda:

```
  javac -sourcepath surse surse/Matematica.java -d clase
sau
  javac -classpath surse surse/Matematica.java -d clase
```

Opţiunea **-d** specifică directorul rădăcină al ierarhiei de clase. In lipsa lui, fiecare unitate de compilare va fi plasată în acelaşi director cu fişierul său sursă.
Deoarece compilăm clasa principală a plicaţiei, vor fi compilate în cascadă toate clasele referite de aceasta, dar numai acestea. In cazul în care dorim să compilăm explicit toate fişierele java dintr-un anumit director, de exemplu `surse/geometrie/plan`, putem folosi expresia:

```
  javac surse/geometrie/plan/*.java -d clase
```

6.2.3 Necesitatea organizării fişierelor

Organizarea fişierelor sursă este necesară deoarece în momentul când compilatorul întâlneste un nume de clasă el trebuie să poată identifica acea clasă, ceea ce înseamna că trebuie să gasească fişerul sursă care o conţine.

Similar, unităţile de compilare sunt organizate astfel pentru a da posibilitatea interpretorului să gasească şi să încarce în memorie o anumită clasă în timpul execuţiei programului.

Insă această organizare nu este suficientă deoarece specifică numai partea finală din calea către fişierele .java şi .class, de exemplu `/matematica/clase/geometrie/plan/Poligon.class`. Pentru aceasta, atât la compilare cât şi la interpretare trebuie specificată lista de directoare rădăcină

în care se găsesc fişierele aplicaţiei. Această listă se numeşte *cale de cautare (classpath)*.

Definiţie
O *cale de căutare* este o listă de directoare sau arhive în care vor fi căutate fişierele necesare unei aplicaţii. Fiecare director din calea de cautare este directorul imediat superior structurii de directoare corespunzătoare numelor pachetelor în care se găsesc clasele din directorul respectiv, astfel încât compilatorul şi interpretorul să poată construi calea completă spre clasele aplicaţiei. Implicit, calea de căutare este formată doar din directorul curent.

Să considerăm clasa principală a aplicaţiei `Matematica.java`:

```
import geometrie.plan.*;
import algebra.Grup;
import analiza.Functie;

public class Matematica {
  public static void main(String args[]) {
    Poligon a = new Poligon();
    geometrie.spatiu.Sfera = new geometrie.spatiu.Sfera();
    //...
  }
}
```

Identificarea unei clase referite în program se face în felul următor:

- La directoarele aflate în calea de căutare se adaugă subdirectoarele specificate în import sau în numele lung al clasei
- In directoarele formate este căutat un fişier cu numele clasei. In cazul în care nu este găsit nici unul sau sunt găsite mai multe va fi semnalată o eroare.

6.2.4 Setarea căii de căutare (CLASSPATH)

Setarea căii de căutare se poate face în două modalităţi:

- Setarea variabilei de mediu **CLASSPATH** - folosind această variantă toate aplicaţiile Java de pe maşina respectivă vor căuta clasele necesare în directoarele specificate în variabila `CLASSPATH`.

```
UNIX:
  SET CLASSPATH = cale1:cale2:...
DOS shell (Windows 95/NT/...):
  SET CLASSPATH = cale1;cale2;...
```

- Folosirea opţiunii **-classpath** la compilarea şi interpretarea programelor - directoarele specificate astfel vor fi valabile doar pentru comanda curentă:

```
javac - classpath <cale de cautare> <surse java>
java  - classpath <cale de cautare> <clasa principala>
```

Lansarea în execuţie a aplicatiei noastre, din directorul `matematica`, se va face astfel:

```
java -classpath clase Matematica
```

In concluzie, o organizare eficientă a fişierelor aplicaţiei ar arăta astfel:

```
/matematica
  /surse
  /clase
  compile.bat
    (javac -sourcepath surse surse/Matematica.java -d clase)
  run.bat
    (java  -classpath clase Matematica)
```

6.3 Arhive JAR

Fişierele **JAR** (Java Archive) sunt arhive în format ZIP folosite pentru împachetarea aplicaţiilor Java. Ele pot fi folosite şi pentru comprimări obişnuite, diferenţa faţă de o arhivă ZIP obişnuită fiind doar existenţa unui director denumit `META-INF`, ce conţine diverse informaţii auxiliare legate de aplicaţia sau clasele arhivate.

Un fişier JAR poate fi creat folosind utilitarul **jar** aflat în distribuţia J2SDK, sau metode ale claselor suport din pachetul **java.util.jar**.

Dintre beneficiile oferite de arhivele JAR amintim:

- portabilitate - este un format de arhivare independent de platformă;

- compresare - dimensiunea unei aplicaţii în forma sa finală este redusă;
- minimizarea timpului de încarcare a unui applet: dacă appletul (fişiere class, resurse, etc) este compresat într-o arhivă JAR, el poate fi încărcat într-o singură tranzacţie HTTP, fără a fi deci nevoie de a deschide câte o conexiune nouă pentru fiecare fişier;
- securitate - arhivele JAR pot fi "semnate" electronic
- mecanismul pentru lucrul cu fişiere JAR este parte integrata a platformei Java.

6.3.1 Folosirea utilitarului jar

Arhivatorul *jar* se găseşte în subdirectorul *bin* al directorului în care este instalat kitul J2SDK. Mai jos sunt prezentate pe scurt operaţiile uzuale:

- Crearea unei arhive
  `jar cf arhiva.jar fişier(e)-intrare`
- Vizualizare conţinutului
  `jar tf nume-arhiva`
- Extragerea conţinutului
  `jar xf arhiva.jar`
- Extragerea doar a unor fişiere
  `jar xf arhiva.jar fişier(e)-arhivate`
- Executarea unei aplicaţii
  `java -jar arhiva.jar`
- Deschiderea unui applet arhivat
  `<applet code=A.class archive="arhiva.jar" ...>`

Exemple:

- Arhivarea a două fişiere class:
  `jar cf classes.jar A.class B.class`
- arhivarea tuturor fişierelor din directorul curent:
  `jar cvf allfiles.jar *`

6.3.2 Executarea aplicaţiilor arhivate

Pentru a rula o aplicaţie împachetată într-o arhivă JAR trebuie să facem cunoscută interpretorului numele clasei principale a aplicaţiei. Să considerăm următorul exemplu, în care dorim să arhivăm clasele aplicaţiei descrise mai sus, în care clasa principală era `Matematica.java`. Din directorul `clase` vom lansa comanda:

```
jar cvf mate.jar geometrie analiza algebra Matematica.class
```

In urma acestei comenzi vom obtine arhiva `mate.jar`. Dacă vom încerca să lansăm în execuţie această arhivă prin comanda `java -jar mate.jar` vom obţine următoarea eroare: "Failed to load Main-Class manifest from mate.jar". Aceasta înseamna că în fiserul `Manifest.mf` ce se gaseşte în directorul `META-INF` trebuie să înregistrăm clasa principală a aplicaţiei. Acest lucru îl vom face în doi paşi:

- se creează un fişier cu un nume oarecare, de exemplu `manifest.txt`, în care vom scrie:
  `Main-Class: Matematica`
- adaugăm această informaţie la fişierul manifest al arhivei mate.jar:
  `jar uvfm mate.jar manifest.txt`

Ambele operaţii puteau fi executate într-un singur pas:

```
jar cvfm mate.jar manifest.txt
    geometrie analiza algebra Matematica.class
```

Pe sistemele Win32, platforma Java 2 va asocia extensiile .jar cu interpretorul Java, ceea ce înseamnă că facând dublu-click pe o arhivă JAR va fi lansată în execuţie aplicaţia împachetată în acea arhivă (dacă există o clasă principală).

Capitolul 7

Colecţii

7.1 Introducere

O *colecţie* este un obiect care grupează mai multe elemente într-o singură unitate. Prin intermediul colecţiilor vom avea acces la diferite tipuri de date cum ar fi vectori, liste înlănţuite, stive, mulţimi matematice, tabele de dispersie, etc. Colecţiile sunt folosite atât pentru memorarea şi manipularea datelor, cât şi pentru transmiterea unor informaţii de la o metodă la alta.

Tipul de date al elementelor dintr-o colecţie este `Object`, ceea ce înseamnă că mulţimile reprezentate sunt eterogene, putând include obiecte de orice tip.

Incepând cu versiunea 1.2, în Java colecţiile sunt tratate într-o manieră unitară, fiind organizate într-o arhitectură foarte eficientă şi flexibilă ce cuprinde:

- **Interfeţe**: tipuri abstracte de date ce descriu colecţiile şi permit utilizarea lor independent de detaliile implementărilor.
- **Implementări**: implementări concrete ale interfeţelor ce descriu colecţii. Aceste clase reprezintă *tipuri de date reutilizabile*.
- **Algoritmi**: metode care efectuează diverse operaţii utile cum ar fi căutarea sau sortarea, definite pentru obiecte ce implementează interfeţele ce descriu colecţii. Aceşti algoritmi se numesc şi *polimorfici* deoarece pot fi folosiţi pe implementări diferite ale unei colecţii, reprezentând elementul de *funcţionalitate reutilizabilă*.

Utilizarea colecţiilor oferă avantaje evidente în procesul de dezvoltare a unei aplicaţii. Cele mai importante sunt:

- **Reducerea efortului de programare**: prin punerea la dispoziţia programatorului a unui set de tipuri de date şi algoritmi ce modelează structuri şi operaţii des folosite în aplicaţii.
- **Creşterea vitezei şi calităţii programului**: implementările efective ale colecţiilor sunt de înaltă performanţă şi folosesc algoritmi cu timp de lucru optim.

Astfel, la scrierea unei aplicaţii putem să ne concentrăm eforturile asupra problemei în sine şi nu asupra modului de reprezentare şi manipulare a informaţiilor.

7.2 Interfeţe ce descriu colecţii

Interfeţele reprezintă nucleul mecanismului de lucru cu colecţii, scopul lor fiind de a permite utilizarea structurilor de date independent de modul lor de implementare.

Collection modelează o colecţie la nivelul cel mai general, descriind un grup de obiecte numite şi *elementele* sale. Unele implementări ale acestei interfeţe permit existenţa elementelor duplicate, alte implementări nu. Unele au elementele ordonate, altele nu. Platforma Java nu oferă nici o implementare directă a acestei interfeţe, ci există doar implementări ale unor subinterfeţe mai concrete, cum ar fi `Set` sau `List`.

```
public interface Collection {

  // Metode cu caracter general
  int size();
  boolean isEmpty();
  void clear();
  Iterator iterator();

  // Operatii la nivel de element
  boolean contains(Object element);
  boolean add(Object element);
  boolean remove(Object element);
```

```
  // Operatii la nivel de multime
  boolean containsAll(Collection c);
  boolean addAll(Collection c);
  boolean removeAll(Collection c);
  boolean retainAll(Collection c);

  // Metode de conversie in vector
  Object[] toArray();
  Object[] toArray(Object a[]);
}
```

Set modelează noţiunea de *mulţime* în sens matematic. O mulţime nu poate avea elemente duplicate, mai bine zis nu poate conţine două obiecte $o1$ şi $o2$ cu proprietatea $o1.equals(o2)$. Moşteneşte metodele din `Collection`, fără a avea alte metode specifice.
Două dintre clasele standard care oferă implementări concrete ale acestei interfeţe sunt **HashSet** şi **TreeSet**.

SortedSet este asemănătoare cu interfaţa `Set`, diferenţa principală constând în faptul că elementele dintr-o astfel de colecţie sunt ordonate ascendent. Pune la dispoziţie operaţii care beneficiază de avantajul ordonării elementelor. Ordonarea elementelor se face conform ordinii lor naturale, sau conform cu ordinea dată de un comparator specificat la crearea colecţiei şi este menţinută automat la orice operaţie efectuată asupra mulţimii. Singura conditţie este ca, pentru orice două obiecte $o1, o2$ ale colecţiei, apelul $o1.compareTo(o2)$ (sau $comparator.compare(o1, o2)$, dacă este folosit un comparator) trebuie să fie valid şi să nu provoace excepţii.

Fiind subclasă a interfeţei `Set`, moşteneşte metodele acesteia, oferind metode suplimentare ce ţin cont de faptul că mulţimea este sortată:

```
public interface SortedSet extends Set {

  // Subliste
  SortedSet subSet(Object fromElement, Object toElement);
  SortedSet headSet(Object toElement);
```

```
  SortedSet tailSet(Object fromElement);

  // Capete
  Object first();
  Object last();

  Comparator comparator();
}
```

Clasa care implementează această interfaţă este **TreeSet**.

List descrie *liste (secvenţe)* de elemente indexate. Listele pot contine duplicate şi permit un control precis asupra poziţiei unui element prin intermediul indexului acelui element. In plus, faţăde metodele definite de interfaţa `Collection`, avem metode pentru acces poziţional, căutare şi iterare avansată. Definiţia interfeţei este:

```
public interface List extends Collection {

  // Acces pozitional
  Object get(int index);
  Object set(int index, Object element);
  void add(int index, Object element);
  Object remove(int index);
  abstract boolean addAll(int index, Collection c);

  // Cautare
  int indexOf(Object o);
  int lastIndexOf(Object o);

  // Iterare
  ListIterator listIterator();
  ListIterator listIterator(int index);

  // Extragere sublista
  List subList(int from, int to);
}
```

Clase standard care implementează această interfaţă sunt: `ArrayList`, `LinkedList`, `Vector`.

Map descrie structuri de date ce asociază fiecarui element o cheie unică, după care poate fi regăsit. Obiectele de acest tip nu pot conţine chei duplicate şi fiecare cheie este asociată la un singur element. Ierarhia interfeţelor derivate din `Map` este independentă de ierarhia derivată din `Collection`. Definitţai interfeţei este prezentată mai jos:

```
public interface Map {
  // Metode cu caracter general
  int size();
  boolean isEmpty();
  void clear();

  // Operatii la nivel de element
  Object put(Object key, Object value);
  Object get(Object key);
  Object remove(Object key);
  boolean containsKey(Object key);
  boolean containsValue(Object value);

  // Operatii la nivel de multime
  void putAll(Map t);

  // Vizualizari ale colectiei
  public Set keySet();
  public Collection values();
  public Set entrySet();

  // Interfata pentru manipularea unei inregistrari
  public interface Entry {
    Object getKey();
    Object getValue();
    Object setValue(Object value);
  }
}
```

Clase care implementează interfaţă `Map` sunt **HashMap**, **TreeMap** şi **Hashtable**.

SortedMap este asemănătoare cu interfaţa `Map`, la care se adaugă faptul că mulţimea cheilor dintr-o astfel de colecţie este menţinută ordonată ascendent conform ordinii naturale, sau conform cu ordinea dată de un comparator specificat la crearea colecţiei. Este subclasa a interfeţei `Map`, oferind metode suplimentare pentru: extragere de subtabele, aflarea primei/ultimei chei, aflarea comparatorului folosit pentru ordonare.
Definiţia interfeţei este dată mai jos:

```
  public interface SortedMap extends Map {

  // Extragerea de subtabele
  SortedMap subMap(Object fromKey, Object toKey);
  SortedMap headMap(Object toKey);
  SortedMap tailMap(Object fromKey);

  // Capete
  Object first();
  Object last();

  // Comparatorul folosit pentru ordonare
  Comparator comparator();
}
```

Clasa care implementează această interfaţă este **TreeMap**.

7.3 Implementări ale colecţiilor

Inainte de versiunea 1.2, exista un set de clase pentru lucrul cu colecţii, însă acestea nu erau organizate pe ierarhia de interfeţe prezentată în secţiunea anterioară. Aceste clase sunt în continuare disponibile şi multe dintre ele au fost adaptate în aşa fel încât să se integreze în noua abordare. Pe lângă acestea au fost create noi clase corespunzătoare interfeţelor definite, chiar dacă funcţionalitatea lor era aproape identică cu cea a unei clase anterioare.

Clasele de bază care implementează interfeţe ce descriu colecţii au numele de forma $< Implementare >< Interfata >$, unde 'implementare' se referă la structura internă folosită pentru reprezentarea mulţimii, şi sunt prezentate în tabelul de mai jos, împreună cu interfeţele corespunzătoare (clasele din vechiul model sunt trecute pe rândul de jos):

Interfaţa	**Clasa**
`Set`	`HashSet`
`SortedSet`	`TreeSet`
`List`	`ArrayList, LinkedList` `Vector`
`Map`	`HashMap` `Hashtable`
`SortedMap`	`TreeMap`

Aşadar se observă existenţa unor clase care oferă aceeaşi funcţionalite, cum ar fi `ArrayList` şi `Vector`, `HashMap` şi `Hashtable`.

Pe lângă organizarea ierarhică a interfeţelor implementate, clasele ce descriu colecţii sunt de asemenea concepute într-o manieră ierarhică, ca în figura de mai jos:

```
AbstractCollection - AbstractSet, AbstractList - HashSet,
TreeSet... Vector-Stack
AbstractMap - HashMap, TreeMap, HashTable
In vechea ierarhie:
Dictionary - Hashtable - Properties
```

Evident, implementarea interfeţelor este explicit realizată la nivelul superclaselor abstracte, acestea oferind de altfel şi implementări concrete pentru multe din metodele definite de interfeţe.

In general, clasele care descriu colecţii au unele trăsaturi comune, cum ar fi:

- permit elementul `null`,
- sunt serializabile,
- au definită metoda `clone`,

- au definită metoda `toString`, care returnează o reprezentare ca şir de caractere a colecţiei respective,
- permit crearea de iteratori pentru parcurgere,
- au atât constructor fără argumente cât şi un constructor care acceptă ca argument o altă colecţie
- exceptând clasele din arhitectura veche, nu sunt sincronizate (vezi "Fire de execuţie).

7.4 Folosirea eficientă a colecţiilor

După cum am vazut, fiecare interfaţă ce descrie o colecţie are mai multe implementări. De exemplu, interfaţa `List` este implementată de clasele `ArrayList` şi `LinkedList`, prima fiind în general mult mai folosită. De ce există atunci şi clasa `LinkedList` ? Raspunsul constă în faptul că folosind reprezentări diferite ale mulţimii gestionate putem obţine performante mai bune în funcţie de situaţie, prin realizarea unor compromisuri între spaţiul necesar pentru memorarea datelor, rapiditatea regăsirii acestora şi timpul necesar actualizării colecţiei în cazul unor modificări.

Să considerăm un exemplu ce creeaza o listă folosind `ArrayList`, respectiv `LinkedList` şi execută diverse operaţii pe ea, cronometrând timpul necesar realizării acestora:

Listing 7.1: Comparare ArrayList - LinkedList

```
import java.util.*;

public class TestEficienta {

  final static int N = 100000;

  public static void testAdd(List lst) {
      long t1 = System.currentTimeMillis();
      for(int i=0; i < N; i++)
        lst.add(new Integer(i));
      long t2 = System.currentTimeMillis();
      System.out.println("Add: " + (t2 - t1));
  }
```

```
  public static void testGet(List lst) {
      long t1 = System.currentTimeMillis();
      for(int i=0; i < N; i++)
        lst.get(i);
      long t2 = System.currentTimeMillis();
      System.out.println("Get: " + (t2 - t1));
  }

  public static void testRemove(List lst) {
      long t1 = System.currentTimeMillis();
      for(int i=0; i < N; i++)
        lst.remove(0);
      long t2 = System.currentTimeMillis();
      System.out.println("Remove: " + (t2 - t1));
  }

  public static void main(String args[]) {
    System.out.println("ArrayList");
    List lst1 = new ArrayList();
    testAdd(lst1);
    testGet(lst1);
    testRemove(lst1);

    System.out.println("LinkedList");
    List lst2 = new LinkedList();
    testAdd(lst2);
    testGet(lst2);
    testRemove(lst2);
  }
}
```

Timpii aproximativi de rulare pe un calculator cu performanţe medii, exprimaţi în secunde, sunt daţi în tabelul de mai jos:

	ArrayList	**LinkedList**
add	0.12	0.14
get	0.01	87.45
remove	12.05	0.01

Aşadar, adăugarea elementelor este rapidă pentru ambele tipuri de liste. `ArrayList` oferă acces în timp constant la elementele sale şi din acest motiv folosirea lui "get" este rapidă, în timp ce pentru `LinkedList` este extrem de lentă, deoarece într-o listă înlanţuită accesul la un element se face prin

parcurgerea secvenţială a listei până la elementul respectiv.
La operaţiunea de eliminare, folosirea lui `ArrayList` este lentă deoarece elementele rămase suferă un proces de reindexare (shift la stânga), în timp ce pentru `LinkedList` este rapidă şi se face prin simpla schimbare a unei legături. Deci, `ArrayList` se comportă bine pentru cazuri în care avem nevoie de regăsirea unor elemente la poziţii diferite în listă, iar `LinkedList` funcţioneaza eficient atunci când facem multe operaţii de modificare (ştergeri, inserări).

Concluzia nu este că una din aceste clase este mai "bună" decât cealaltă, ci că există diferenţe substanţiale în reprezentarea şi comportamentul diferitelor implementări şi că alegerea unei anumite clase pentru reprezentarea unei mulţimi de elemente trebuie să se facă în funcţie de natura problemei ce trebuie rezolvată.

7.5 Algoritmi polimorfici

Algoritmii polimorfici descrişi în această secţiune sunt metode definite în clasa **Collections** care permit efectuarea unor operaţii utile cum ar fi căutarea, sortarea, etc. Caracterisiticile principale ale acestor algoritmi sunt:

- sunt metode de clasă (statice);
- au un singur argument de tip colecţie;
- apelul lor general va fi de forma:
  `Collections.algoritm(colectie, [argumente])`;
- majoritatea operează pe liste dar şi pe colecţii arbitrare.

Metodele mai des folosite din clasa `Collections` sunt:

- **sort** - sortează ascendent o listă referitor la ordinea să naturală sau la ordinea dată de un comparator;
- **shuffle** - amestecă elementele unei liste - opusul lui `sort`;
- **binarySearch** - efectuează căutarea eficientă (binară) a unui element într-o listă ordonată;

- **reverse** - inversează ordinea elementelor dintr-o listă;
- **fill** - populeaza o lista cu un anumit element repetat de un număr de ori;
- **copy** - copie elementele unei liste în alta;
- **min** - returnează minimul dintr-o colecţie;
- **max** - returnează maximul dintr-o colecţie;
- **swap** - interschimbă elementele de la două poziţii specificate ale unei liste;
- **enumeration** - returneaza o enumerare a elementelor dintr-o colecţie;
- **unmodifiable***TipColectie* - returnează o instanţă care nu poate fi modificată a colecţiei respective;
- **synchronized***TipColectie* - returnează o instanţă sincronizată a unei colecţii (vezi "Fire de execuţie").

7.6 Tipuri generice

Tipurile generice, introduse în versiunea 1.5 a limbajului Java, simplifică lucrul cu colecţii, permiţând tipizarea elementelor acestora. Definirea unui tip generic se realizează prin specificarea între paranteze unghiulare a unui tip de date Java, efectul fiind impunerea tipului respectiv pentru toate elementele colecţiei: <**TipDate**>. Să considerăm un exemplu de utilizare a colecţiilor înainte şi după introducerea tipurilor generice:

```
// Inainte de 1.5
ArrayList list = new ArrayList();
list.add(new Integer(123));
int val = ((Integer)list.get(0)).intValue();
```

In exemplul de mai sus, lista definită poate conţine obiecte de orice tip, deşi am dori ca elementele să fie doar numere întregi. Mai mult, trebuie să facem cast explicit de la tipul `Object` la `Integer` atunci când preluăm valoarea unui element.

Folosind tipuri generice, putem rescrie secvenţa astfel:

```
// Dupa 1.5, folosind tipuri generice
ArrayList<Integer> list =  new ArrayList<Integer>();
list.add(new Integer(123));
int val = list.get(0).intValue();
```

Dacă utilizăm şi mecanismul de *autoboxing*, obţinem o variantă mult simplificată a secvenţei iniţiale:

```
// Dupa 1.5, folosind si autoboxing
ArrayList<Integer> list =  new ArrayList<Integer>();
list.add(123);
int val = list.get(0);
```

In cazul folosirii tipurilor generice, încercarea de a utiliza în cadrul unei colecţii a unui element necorespunzător ca tip va produce o eroare la compilare, spre deosebire de varianta anterioară ce permitea doara aruncarea unor excepţie de tipul `ClassCastException` în cazul folosirii incorecte a tipurilor.

7.7 Iteratori şi enumerări

Enumerările şi iteratorii descriu modalităţi pentru parcurgerea secvenţială a unei colecţii, indiferent dacă aceasta este indexată sau nu. Ei sunt descrişi de obiecte ce implementează interfeţele **Enumeration**, respectiv **Iterator** sau **ListIterator**. Toate clasele care implementează colecţii au metode ce returnează o enumerare sau un iterator pentru parcurgerea elementelor lor. Deoarece funcţionalitatea interfeţei `Enumeration` se regăseşte în `Iterator`, aceasta din urmă este preferată în noile implementări ale colecţiilor.

Metodele uzuale ale acestor interfeţe sunt prezentate mai jos, împreună cu modalitatea lor de folosire, semnificaţiile lor fiind evidente:

- **Enumeration**: `hasMoreElements, nextElement`

  ```
  // Parcurgerea elementelor unui vector v
  Enumeration e = v.elements;
  while (e.hasMoreElements()) {
    System.out.println(e.nextElement());
  }
  // sau, varianta mai concisa
  for (Enumeration e = v.elements();
  ```

```
        e.hasMoreElements();) {
  System.out.println(e.nextElement());
}
```

- **Iterator**: `hasNext, next, remove`

```
// Parcurgerea elementelor unui vector
// si eliminarea elementelor nule
for (Iterator it = v.iterator(); it.hasNext();) {
  Object obj = it.next();
  if (obj == null)
    it.remove();
}
```

- **ListIterator**: `hasNext, hasPrevious, next, previous, remove, add, set`

```
// Parcurgerea elementelor unui vector
// si inlocuirea elementelor nule cu 0
for (ListIterator it = v.listIterator();
       it.hasNext();) {
  Object obj = it.next();
  if (obj == null)
    it.set(new Integer(0));
}
```

Iteratorii simpli permit eliminarea elementului curent din colecţia pe care o parcurg, cei de tip `ListIterator` permit şi inserarea unui element la poziţia curentă, respectiv modificarea elementului curent, precum şi iterarea în ambele sensuri. Iteratorii sunt preferaţi enumerărilor datorită posibilităţii lor de a acţiona asupra colecţiei pe care o parcurg prin metode de tip `remove, add, set` dar şi prin faptul că denumirile metodelor sunt mai concise.

Atenţie

Deoarece colecţiile sunt construite peste tipul de date `Object`, metodele de tip `next` sau `prev` ale iteratorilor vor returna tipul `Object`, fiind responsabilitatea noastră de a face conversie (cast) la alte tipuri de date, dacă este cazul.

In exemplul de mai jos punem într-un vector numerele de la 1 la 10, le amestecăm, după care le parcurgem element cu element folosind un iterator, înlocuind numerele pare cu 0.

Listing 7.2: Folosirea unui iterator

```
import java.util.*;
class TestIterator {
  public static void main(String args[]) {
    ArrayList a = new ArrayList();

    // Adaugam numerele de la 1 la 10
    for(int i=1; i<=10; i++)
      a.add(new Integer(i));

    // Amestecam elementele colectiei
    Collections.shuffle(a);
    System.out.println("Vectorul amestecat: " + a);

    // Parcurgem vectorul
    for(ListIterator it=a.listIterator(); it.hasNext(); ) {
      Integer x = (Integer) it.next();

      // Daca elementul curent este par, il facem 0
      if (x.intValue() % 2 == 0)
        it.set(new Integer(0));
    }
    System.out.print("Rezultat: " + a);
  }
}
```

Incepând cu versiunea 1.5 a limbajului Java, există o variantă simplificată de utilizare a iteratorilor. Astfel, o secvenţă de genul:

```
ArrayList<Integer> list = new ArrayList<Integer>();
for (Iterator i = list.iterator(); i.hasNext();) {
  Integer val=(Integer)i.next();
  // Proceseaza val
  ...
```

```
}
```

poate fi rescrisă astfel:

```
ArrayList<Integer> list = new ArrayList<Integer>();
for (Integer val : list) {
  // Proceseaza val
  ...
}
```

Capitolul 8

Serializarea obiectelor

8.1 Folosirea serializării

Definiţie
Serializarea este o metodă ce permite transformarea unui obiect într-o secvenţă de octeţi sau caractere din care să poată fi refăcut ulterior obiectul original.

Cu alte cuvinte, serializarea permite salvarea într-o manieră unitară a tuturor informaţiilor unui obiect pe un mediu de stocare extern programului. Procesul invers, de citire a unui obiect serializat pentru a-i reface starea originală, se numeşte *deserializare*. Intr-un cadru mai larg, prin serializare se întelege procesul de scriere/citire a obiectelor.

Tipurile primitive pot fi de asemenea serializate.

Utilitatea serializarii constă în următoarele aspecte:

- Asigură un mecanism *simplu de utilizat* pentru salvarea şi restaurarea a datelor.

- Permite *persistenţa obiectelor*, ceea ce înseamna că durata de viaţa a unui obiect nu este determinată de execuţia unui program în care acesta este definit - obiectul poate exista şi între apelurile programelor care îl folosesc. Acest lucru se realizează prin serializarea obiectului şi scrierea lui pe disc înainte de terminarea unui program, apoi, la relansarea programului, obiectul va fi citit de pe disc şi starea lui refacută. Acest

tip de persistenţă a obiectelor se numeşte *persistenţă uşoară*, întrucât ea trebuie efectuată explicit de către programator şi nu este realizată automat de către sistem.

- *Compensarea diferenţelor între sisteme de operare* - transmiterea unor informaţii între platforme de lucru diferite se realizează unitar, independent de formatul de reprezentare a datelor, ordinea octeţilor sau alte detalii specifice sistemelor repective.

- *Transmiterea datelor în reţea* - Aplicaţiile ce rulează în reţea pot comunica între ele folosind fluxuri pe care sunt trimise, respectiv recepţionate obiecte serializate.

- *RMI (Remote Method Invocation)* - este o modalitate prin care metodele unor obiecte de pe o altă maşină pot fi apelate ca şi cum acestea ar exista local pe maşina pe care rulează aplicaţia. Atunci când este trimis un mesaj către un obiect "remote" (de pe altă maşină), serializarea este utilizată pentru transportul argumentelor prin reţea şi pentru returnarea valorilor.

- *Java Beans* - sunt componente reutilizabile, de sine stătătoare ce pot fi utilizate în medii vizuale de dezvoltare a aplicaţiilor. Orice componentă Bean are o stare definită de valorile implicite ale proprietăţilor sale, stare care este specificată în etapa de design a aplicaţiei. Mediile vizuale folosesc mecanismul serializării pentru asigurarea persistenţei componentelor Bean.

Un aspect important al serializării este că nu salvează doar imaginea unui obiect ci şi toate referinţele la alte obiecte pe care acesta le conţine. Acesta este un proces recusiv de salvare a datelor, întrucât celelalte obiectele referite de obiectul care se serializează pot referi la rândul lor alte obiecte, şi aşa mai departe. Aşadar referinţele care construiesc starea unui obiect formează o întreagă reţea, ceea ce înseamnă că un algoritm general de salvare a stării unui obiect nu este tocmai facil.

In cazul în care starea unui obiect este formată doar din valori ale unor variabile de tip primitiv, atunci salvarea informaţiilor încapsulate în acel obiect se poate face şi prin salvarea pe rând a datelor, folosind clasa `DataOutputStream`, pentru ca apoi să fie restaurate prin metode ale clasei `DataInputStream`, dar, aşa cum am vazut, o asemenea abordare nu este

în general suficientă, deoarece pot apărea probleme cum ar fi: variabilele membre ale obiectului pot fi instanţe ale altor obiecte, unele câmpuri pot face referinţă la acelaşi obiect, etc.

Serializarea în format binar a tipurilor primitive şi a obiectelor se realizează prin intermediul fluxurilor definite de clase specializate în acest scop cu ar fi:
`ObjectOutputStream` pentru scriere şi `ObjectInputStream` pentru restaurare.

In continuare, prin termenul *serializare* ne vom referi doar la serializarea în format binar.

8.1.1 Serializarea tipurilor primitive

Serializarea tipurilor primitive poate fi realizată fie prin intermediul fluxurilor `DataOutputStream` şi `DataInputStream`, fie cu `ObjectOutputStream` şi `ObjectInputStream`. Acestea implementează interfeţele `DataInput`, respectiv `DataOutput` ce declară metode de tipul `read`*TipPrimitiv*, respectiv `write`*TipPrimitiv* pentru scrierea/citirea datelor primitive şi a şirurilor de caractere.

Mai jos este prezentat un exemplu de serializare folosind clasa `DataOutputStream`:

```
FileOutputStream fos = new FileOutputStream("test.dat");
DataOutputStream out = new DataOutputStream(fos);
out.writeInt(12345);
out.writeDouble(12.345);
out.writeBoolean(true);
out.writeUTF("Sir de caractere");
out.flush();
fos.close();
```

Citirea informaţiilor scrise în exemplul de mai sus se va face astfel:

```
FileInputStream fis = new FileInputStream("test.dat");
DataInputStream in = new DataInputStream(fis);
int i = in.readInt();
double d = in.readDouble();
boolean b = in.readBoolean();
```

```
String s = in.readUTF();
fis.close();
```

8.1.2 Serializarea obiectelor

Serializarea obiectelor se realizează prin intermediul fluxurilor definite de clasele **ObjectOutputStream** (pentru salvare) şi **ObjectInputStream** (pentru restaurare). Acestea sunt fluxuri de procesare, ceea ce înseamna că vor fi folosite împreuna cu alte fluxuri pentru scrierea/citirea efectivă a datelor pe mediul extern pe care va fi salvat, sau de pe care va fi restaurat un obiect serializat.

Mecanismul implicit de serializare a unui obiect va salva numele clasei obiectului, signatura clasei şi valorile tuturor câmpurile serializabile ale obiectului. Referinţele la alte obiecte serializabile din cadrul obiectului curent vor duce automat la serializarea acestora iar referinţele multiple către un acelaşi obiect sunt codificate utilizând un algoritm care să poată reface "reţeaua de obiecte" la aceeaşi stare ca atunci când obiectul original a fost salvat.

Clasele `ObjectInputStream` şi `ObjectOutputStream` implementează interfeţele `ObjectInput`, respectiv `ObjectOutput` care extind `DataInput`, respectiv `DataOutput`, ceea ce înseamnă că, pe lângă metodele dedicate serializării obiectelor, vor exista şi metode pentru scrierea/citirea datelor primitive şi a şirurilor de caractere.

Metodele pentru serializarea obiectelor sunt:

- **writeObject**, pentru scriere şi
- **readObject**, pentru restaurare.

8.1.3 Clasa ObjectOutputStream

Scrierea obiectelor pe un flux de ieşire este un proces extrem de simplu, secvenţa uzuală fiind cea de mai jos:

```
ObjectOutputStream out = new ObjectOutputStream(fluxPrimitiv);
out.writeObject(referintaObiect);
out.flush();
fluxPrimitiv.close();
```

Exemplul de mai jos construieşte un obiect de tip `Date` şi îl salvează în fişierul `test.ser`, împreună cu un obiect de tip `String`. Evident, fişierul rezultat va conţine informaţiile reprezentate în format binar.

```
FileOutputStream fos = new FileOutputStream("test.ser");
ObjectOutputStream out = new ObjectOutputStream(fos);
out.writeObject("Ora curenta:");
out.writeObject(new Date());
out.flush();
fos.close();
```

Deoarece implementează interfaţa `DataOutput`, pe lânga metoda de scriere a obiectelor, clasa pune la dispozitie şi metode de tipul `write`*TipPrimitiv* pentru serializarea tipurilor de date primitive şi a şirurilor de caractere, astfel încât apeluri ca cele de mai jos sunt permise :

```
out.writeInt(12345);
out.writeDouble(12.345);
out.writeBoolean(true);
out.writeUTF("Sir de caractere");
```

Metoda `writeObject` aruncă excepţii de tipul `IOException` şi derivate din aceasta, mai precis `NotSerializableException` dacă obiectul primit ca argument nu este serializabil, sau `InvalidClassException` dacă sunt probleme cu o clasă necesară în procesul de serializare. Vom vedea în continuare că un obiect este serializabil dacă este instanţă a unei clase ce implementează interfaţa `Serializable`.

8.1.4 Clasa ObjectInputStream

Odată ce au fost scrise obiecte şi tipuri primitive de date pe un flux, citirea acestora şi reconstruirea obiectelor salvate se va face printr-un flux de intrare de tip `ObjectInputStream`. Acesta este de asemenea un flux de procesare şi va trebui asociat cu un flux pentru citirea efectivă a datelor, cum ar fi `FileInputStream` pentru date salvate într-un fişier. Secvenţa uzuală pentru deserializare este cea de mai jos:

```
ObjectInputStream in = new ObjectInputStream(fluxPrimitiv);
Object obj = in.readObject();
//sau
```

```
TipReferinta ref = (TipReferinta)in.readObject();
fluxPrimitiv.close();
```

Citirea informaţiilor scrise în exemplul de mai sus se va face astfel:

```
FileInputStream fis = new FileInputStream("test.ser");
ObjectInputStream in = new ObjectInputStream(fis);
String mesaj = (String)in.readObject();
Date data = (Date)in.readObject();
fis.close();
```

Trebuie observat că metoda `readObject` are tipul returnat `Object`, ceea ce înseamnă că trebuie realizată explicit conversia la tipul corespunzator obiectului citit:

```
Date date = in.readObject(); // gresit
Date date = (Date)in.readObject(); // corect
```

Atenţie

Ca şi la celelalte fluxuri de date care implemetează interfaţa `DataInput` citirea dintr-un flux de obiecte trebuie să se facă exact în ordinea în carea acestea au fost scrise, altfel vor apărea evident excepţii în procesul de deserializare.

Clasa `ObjectInputStream` implementează interfaţa `DataInput` deci, pe lângă metoda de citire a obiectelor, clasa pune la dispoziţie şi metode de tipul `read`*TipPrimitiv* pentru citirea tipurilor de date primitive şi a şirurilor de caractere.

```
int i = in.readInt();
double d = in.readDouble();
boolean b = in.readBoolean();
String s = in.readUTF();
```

8.2 Obiecte serializabile

Un obiect este serializabil dacă şi numai dacă clasa din care face parte implementează interfaţa **Serializable**. Aşadar, dacă dorim ca instanţele unei clase să poată fi serializate, clasa respectivă trebuie să implementeze, direct sau indirect, interfaţa `Serializable`.

8.2.1 Implementarea interfeţei Serializable

Interfaţa `Serializable` nu conţine nici o declaraţie de metodă sau constantă, singurul ei scop fiind de a identifica clasele ale căror obiecte sunt serializabile. Definiţia sa completă este:

```
package java.io;
public interface Serializable {
  // Nimic !
}
```

Declararea claselor ale căror instanţe trebuie să fie serializate este aşadar extrem de simplă, fiind făcută prin simpla implementare a interfeţei `Serializable`:

```
public class ClasaSerializabila implements Serializable {
  // Corpul clasei
}
```

Orice subclasă a unei clase serializabile este la rândul ei serializabilă, întrucât implementează indirect interfaţa `Serializable`.

In situaţia în care dorim să declarăm o clasă serializabilă dar superclasa sa nu este serializabilă, atunci trebuie să avem în vedere următoarele lucruri:

- Variabilele accesibile ale superclasei nu vor fi serializate, fiind responsabilitatea clasei curente de a asigura un mecanism propriu pentru salvarea/restaurarea lor. Acest lucru va fi discutat în secţiunea referitoare la personalizarea serializării.

- Superclasa trebuie să aibă obligatoriu un constructor accesibil fără argumente, acesta fiind utilizat pentru iniţializarea variabilelor moştenite în procesul de restaurare al unui obiect. Variabilele proprii vor fi iniţializate cu valorile de pe fluxul de intrare. In lipsa unui constructor accesibil fără argumente pentru superclasă, va fi generată o excepţie la execuţie.

In procesul serializării, dacă este întâlnit un obiect care nu implementează interfaţa `Serializable` atunci va fi generată o excepţie de tipul `NotSerializableException` ce va identifica respectiva clasă neserializabilă.

8.2.2 Controlul serializării

Există cazuri când dorim ca unele variabile membre ale unui obiect să nu fie salvate automat în procesul de serializare. Acestea sunt cazuri comune atunci când respectivele câmpuri reprezintă informaţii confidenţiale, cum ar fi parole, sau variabile temporare pe care nu are rost să le salvăm. Chiar declarate private în cadrul clasei aceste câmpuri participă la serializare. Pentru ca un câmp să nu fie salvat în procesul de serializare el trebuie declarat cu modificatorul **transient** şi trebuie să fie ne-static. De exemplu, declararea unei variabile membre temporare ar trebui facută astfel:

```
transient private double temp;
// Ignorata la serializare
```

Modificatorul `static` anulează efectul modificatorului `transient`. Cu alte cuvinte, variabilele de clasă participă obligatoriu la serializare.

```
static transient int N;
// Participa la serializare
```

In exemplele următoare câmpurile marcate 'DA' participă la serializare, cele marcate 'NU', nu participă iar cele marcate cu 'Exceptie' vor provoca excepţii de tipul `NotSerializableException`.

Listing 8.1: Modificatorii static şi transient

```
import java.io.*;

public class Test1 implements Serializable {

  int x=1;                    //DA
  transient int y=2;          //NU
  transient static int z=3;   //DA
  static int t=4;             //DA

  public String toString() {
    return x + ", " + y + ", " + z + ", " + t;
  }
}
```

Dacă un obiect ce trebuie serializat are referinţe la obiecte neserializabile, atunci va fi generată o excepţie de tipul `NotSerializableException`.

Listing 8.2: Membrii neserializabili

```
import java.io.*;

class A {
  int x=1;
}

class B implements Serializable {
  int y=2;
}

public class Test2 implements Serializable{
  A a = new A();  //Exceptie
  B b = new B();  //DA

  public String toString() {
    return a.x + ", " + b.y;
  }
}
```

Atunci când o clasă serializabila deriva dintr-o altă clasă, salvarea câmpurilor clasei părinte se va face doar dacă şi aceasta este serializabilă. In caz contrar, subclasa trebuie să salveze explicit şi câmpurile moştenite.

Listing 8.3: Serializarea câmpurilor moştenite

```
import java.io.*;
class C {
  int x=0;
  // Obligatoriu constructor fara argumente
}

class D extends C implements Serializable {
  int y=0;
}

public class Test3 extends D {
  public Test3() {
    x = 1; //NU
    y = 2; //DA
```

```
  }
  public String toString() {
    return x + ", " + y;
  }
}
```

Mai jos este descrisa o aplicaţie care efectuează salvarea şi restaurarea unor obiecte din cele trei clase prezentate mai sus.

Listing 8.4: Testarea serializării

```
import java.io.*;

public class Exemplu {

  public static void test(Object obj) throws IOException {
    // Salvam
    FileOutputStream fos = new FileOutputStream("fisier.ser")
        ;
    ObjectOutputStream out = new ObjectOutputStream(fos);

    out.writeObject(obj);
    out.flush();

    fos.close();
    System.out.println("A fost salvat obiectul: " + obj);

    // Restauram
    FileInputStream fis = new FileInputStream("fisier.ser");
    ObjectInputStream in = new ObjectInputStream(fis);
    try {
      obj = in.readObject();
    } catch(ClassNotFoundException e) {
      e.printStackTrace();
    }

    fis.close();
    System.out.println("A fost restaurat obiectul: " + obj);
  }

  public static void main(String args[]) throws IOException {
    test(new Test1());
    try {
      test(new Test2());
    } catch(NotSerializableException e) {
```

```
            System.out.println("Obiect neserializabil: " + e);
        }

        test(new Test3());
    }
}
```

Rezultatul acestui program va fi :

```
A fost salvat obiectul: 1, 2, 3, 4
A fost restaurat obiectul: 1, 0, 3, 4
Obiect neserializabil: java.io.NotSerializableException: A
A fost salvat obiectul: 1, 2
A fost restaurat obiectul: 0, 2
```

8.3 Personalizarea serializării obiectelor

Dezavantajul mecanismului implicit de serializare este că algoritmul pe care se beazează, fiind creat pentru cazul general, se poate comporta ineficient în anumite situaţii: poate fi mult mai lent decât este cazul sau reprezentarea binară generată pentru un obiect poate fi mult mai voluminoasă decât ar trebui. In aceste situaţii, putem să înlocuim algoritmul implicit cu unul propriu, particularizat pentru o clasă anume. De asemenea, este posibil să extindem comportamentul implicit, adăugând şi alte informaţii necesare pentru serializarea unor obiecte.
In majoritatea cazurilor mecanismul standard este suficient însă, după cum am spus, o clasă poate avea nevoie de mai mult control asupra serializării.

Personalizarea serializarii se realizează prin definirea (într-o clasă serializabilă!) a metodelor `writeObject` şi `readObject` având **exact** signatura de mai jos:

```
  private void writeObject(java.io.ObjectOutputStream stream)
     throws IOException
  private void readObject(java.io.ObjectInputStream stream)
     throws IOException, ClassNotFoundException
```

Metoda `writeObject` controlează ce date sunt salvate iar `readObject` controlează modul în care sunt restaurate obiectele, citind informaţiile salvate şi, eventual, modifcând starea obiectelor citite astfel încât ele să corespundă

anumitor cerinţe. In cazul în care nu dorim să înlocuim complet mecanismul standard, putem să folosim metodele **defaultWriteObject**, respectiv **defaultReadObject** care descriu procedurile implicite de serializare.

Forma generală de implementare a metodelor `writeObject` şi `readObject` este:

```
private void writeObject(ObjectOutputStream stream)
    throws IOException {

  // Procesarea campurilor clasei (criptare, etc.)
  ...
  // Scrierea obiectului curent
  stream.defaultWriteObject();

  // Adaugarea altor informatii suplimentare
  ...
}

private void readObject(ObjectInputStream stream)
    throws IOException,ClassNotFoundException {

  // Restaurarea obiectului curent
  stream.defaultReadObject();

  // Actualizarea starii obiectului (decriptare, etc.)
  // si extragerea informatiilor suplimentare
  ...
}
```

Metodele `writeObject` şi `readObject` sunt responsabile cu serializarea clasei în care sunt definite, serializarea superclasei sale fiind facută automat (şi implicit). Dacă însă o clasă trebuie sa-şi coordoneze serializarea proprie cu serializarea superclasei sale, atunci trebuie să implementeze interfaţa `Externalizable`.

8.3.1 Controlul versiunilor claselor

Să presupunem că dorim să realizăm o aplicaţie care să ţină evidenţa angajaţilor unei companii. Evident, vom avean nevoie de o clasă care să reprezinte

noţiunea de *angjat*. O variantă simplificată a acesteia ar putea arăta astfel:

Listing 8.5: Prima variantă a clasei Angajat

```
import java.io.*;

class Angajat implements Serializable {

  public String nume;
  public int salariu;
  private String parola;

  public Angajat(String nume, int salariu, String parola) {
    this.nume = nume;
    this.salariu = salariu;
    this.parola = parola;
  }

  public String toString() {
    return nume + " (" + salariu + ")";
  }

}
```

Mai jos este prezentată o mică aplicaţie care permite introducerea de angajaţi şi salvarea lor într-un fişier. La fiecare pornire a aplicaţiei, vor fi citite datele din fişier astfel încât programul va actualiza în permanenţă lista angajaţilor cu noi persoane.

Listing 8.6: Aplicaţia de gestionare a angajaţilor

```
import java.io.*;
import java.util.*;

public class GestiuneAngajati {

  //Lista angajatilor
  ArrayList ang = new ArrayList();

  public void citire() throws IOException {
    FileInputStream fis = null;
    try {
      fis = new FileInputStream("angajati.ser");
      ObjectInputStream in = new ObjectInputStream(fis);
```

```
      ang = (ArrayList) in.readObject();
    } catch(FileNotFoundException e) {
      System.out.println("Fisierul nou...");
    } catch(Exception e) {
      System.out.println("Eroare la citirea datelor...");
      e.printStackTrace();
    }finally {
      if (fis != null)
        fis.close();
    }
    System.out.println("Lista angajatilor:\n" + ang);
  }

  public void salvare() throws IOException {
    FileOutputStream fos =
      new FileOutputStream("angajati.ser");
    ObjectOutputStream out = new ObjectOutputStream(fos);
    out.writeObject(ang);
  }

  public void adaugare() throws IOException {
     BufferedReader stdin = new BufferedReader(
       new InputStreamReader(System.in));

     while (true) {
       System.out.print("\nNume:");
       String nume = stdin.readLine();

       System.out.print("Salariu:");
       int salariu = Integer.parseInt(stdin.readLine());

       System.out.print("Parola:");
       String parola = stdin.readLine();

       ang.add(new Angajat(nume, salariu, parola));

       System.out.print("Mai adaugati ? (D/N)");
       String raspuns = stdin.readLine().toUpperCase();
       if (raspuns.startsWith("N"))
         break;
     }
  }

  public static void main(String args[]) throws IOException {
    GestiuneAngajati app = new GestiuneAngajati();
```

```
    //Incarcam angajatii din fisier
    app.citire();

    //Adaugam noi angajati
    app.adaugare();

    //Salvam angajatii inapoi fisier
    app.salvare();
  }
}
```

Problema care se pune acum este următoarea. După introducerea unui număr suficient de mare de angajaţi în fişier, clasa `Angajat` este modificată prin adăugarea unei noi variabilă membre care să reţina *adresa*. La execuţia aplicaţiei noastre, procedura de citire a angajaţilor din fişier nu va mai funcţiona, producând o excepţie de tipul `InvalidClassException`. Aceastăproblemă ar fi apărut chiar dacă variabila adăugată era declarată de tip `transient`. De ce se întâmplâ acest lucru ?

Explicaţia constă în faptul că mecanismul de serializare Java este foarte atent cu signatura claselor serializate. Pentru fiecare obiect serializat este calculat automat un număr reprezentat pe 64 de biţi, care reprezintă un fel de "amprentă" a clasei obiectului. Acest număr, denumit **serialVersionUID**, este generat pornind de la diverse informaţii ale clasei, cum ar fi variabilele sale membre, (dar nu numai) şi este salvat în procesul de serializare împreună cu celelalte date. In plus, orice modificare semnificativă a clasei, cum ar fi adăugarea unui nou câmp, va determina modificarea numărului său de versiune.

La restaurarea unui obiect, numărul de versiune salvat în forma serializată va fi regăsit şi comparat cu noua semnătură a clasei obiectului. In cazul în care acestea nu sunt egale, va fi generată o excepţie de tipul `InvalidClassException` şi deserializarea nu va fi făcută.

Această abordare extrem de precaută este foarte utilă pentru prevenirea unor anomalii ce pot apărea când două versiuni de clase sunt incompatibile, dat poate fi supărătoare atunci când modificările aduse clasei nu strică compatibilitatea cu vechea versiune. In această situaţie trebuie să comunicăm explicit că cele două clase sunt compatibile. Acest lucru se realizează prin setarea manuală a variabilei `serialVersionUID` în cadrul clasei dorite, adăugând pur şi simplu câmpul:

```
static final long serialVersionUID = /* numar_serial_clasa */;
```

Prezenţa variabilei `serialVersionUID` printre membrii unei clase va informa algoritmul de serialzare că nu mai calculeze numărul de serie al clasei, ci să-l folosească pe cel specificat de noi. Cum putem afla însă numărul de serie al vechii clase `Angajat` care a fost folosită anterior la salvarea angajaţilor ?

Utilitarul **serialVer** permite generarea numărului `serialVersionUID` pentru o clasă specificată. Aşadar, trebuie să recompilăm vechea clasă `Angajat` şi să-i aflăm numărul de serie astfel: `serialVer Angajat`. Rezultatul va fi:

```
Angajat:
static final long serialVersionUID = 5653493248680665297L;
```

Vom rescrie noua clasă `Angajat` astfel încât să fie compatibilă cu cea veche astfel:

Listing 8.7: Variantă compatibilă a clasei Angajat

```
import java.io.*;

class Angajat implements Serializable {

  static final long serialVersionUID = 5653493248680665297L;

  public String nume, adresa;
  public int salariu;
  private String parola;

  public Angajat(String nume, int salariu, String parola) {
    this.nume = nume;
    this.adresa = "Iasi";
    this.salariu = salariu;
    this.parola = parola;
  }

  public String toString() {
    return nume + " (" + salariu + ")";
  }

}
```

Aplicaţia noastră va funcţiona acum, însă rubrica *adresă* nu va fi iniţializată în nici un fel (va fi null), deoarece ea nu exista în formatul original. La noua

salvare a datelor, vor fi serializate şi informaţiile legate de adresă (evident, trebuie însă să le citim de la tastatură...)

8.3.2 Securizarea datelor

După cum am văzut membrii privaţi, cum ar fi *parola* din exemplul de mai sus, participă la serializare. Problema constă în faptul că, deşi în format binar, informaţiile unui obiect serializat nu sunt criptate în nici un fel şi pot fi regăsite cu uşurinţă, ceea ce poate reprezenta un inconvenient atunci când există câmpuri confidenţiale.

Rezolvarea acestei probleme se face prin modificarea mecanismului implicit de serializare, implementând metodele **readObject** şi **writeObject**, precum şi prin utilizarea unei funcţii de criptare a datelor. Varianta securizată a clasei `Angajat` din exemplul anterior ar putea arăta astfel:

Listing 8.8: Varianta securizată a clasei Angajat

```
import java.io.*;

class Angajat implements Serializable {

  static final long serialVersionUID = 5653493248680665297L;

  public String nume, adresa;
  public int salariu;
  private String parola;

  public Angajat(String nume, int salariu, String parola) {
    this.nume = nume;
    this.adresa = "Iasi";
    this.salariu = salariu;
    this.parola = parola;
  }

  public String toString() {
    return nume + " (" + salariu + ")";
  }

  static String criptare(String input, int offset) {
    StringBuffer sb = new StringBuffer();
    for (int n=0; n<input.length(); n++)
      sb.append((char)(offset+input.charAt(n)));
    return sb.toString();
```

```
  }

  private void writeObject(ObjectOutputStream stream)
      throws IOException {
    parola = criptare(parola, 3);
    stream.defaultWriteObject();
    parola = criptare(parola, -3);
  }

  private void readObject(ObjectInputStream stream)
      throws IOException, ClassNotFoundException {
    stream.defaultReadObject();
    parola = criptare(parola, -3);
  }
}
```

8.3.3 Implementarea interfeţei Externalizable

Pentru un control complet, explicit, al procesului de serializare, o clasă trebuie să implementeze interfaţa **Externalizable**. Pentru instanţe ale acestor clase doar numele clasei este salvat automat pe fluxul de obiecte, clasa fiind responsabilă cu scrierea şi citirea membrilor săi şi trebuie să se coordoneze cu superclasele ei.

Definiţia interfeţei `Externalizable` este:

```
package java.io;
public interface Externalizable extends Serializable {
  public void writeExternal(ObjectOutput out)
    throws IOException;
  public void readExternal(ObjectInput in)
    throws IOException, ClassNotFoundException;
}
```

Aşadar, aceste clase trebuie să implementeze obligatoriu metodele **writeExternal** şi **readExternal** în care se va face serializarea completă a obiectelor şi coordonarea cu superclasa ei.

Uzual, interfaţa `Externalizable` este folosită în situaţii în care se doreşte îmbunătăţirea performanţelor algoritmului standard, mai exact creşterea vitezei procesului de serializare.

Mai jos este prezentată o clasă simplă şi modalitatea de rescriere a sa folosind interfaţa `Externalizable`:

Listing 8.9: Serializare implicită

```
import java.io.*;

class Persoana implements Serializable {
  int cod;
  String nume;

  public Persoana(String nume, int cod) {
    this.nume = nume;
    this.cod  = cod;
  }
}
```

Listing 8.10: Serializare proprie

```
import java.io.*;

class Persoana implements Externalizable {
  int cod;
  String nume;

  public Persoana(String nume, int cod) {
    this.nume = nume;
    this.cod  = cod;
  }

  public void writeExternal(ObjectOutput s)
      throws IOException {
    s.writeUTF(nume);
    s.writeInt(cod);
  }

  public void readExternal(ObjectInput s)
      throws ClassNotFoundException, IOException {
    nume = s.readUTF();
    cod = s.readInt();
  }

}
```

8.4 Clonarea obiectelor

Se ştie că nu putem copia valoarea unui obiect prin instrucţiunea de atribuire. O secvenţa de forma:

```
TipReferinta o1 = new TipReferinta();
TipReferinta o2 = o1;
```

nu face decât să declare o nouă variabilă $o2$ ca referinţa la obiectul referit de $o1$ şi nu creează sub nici o formă un nou obiect.

O posibilitate de a face o copie a unui obiect este folosirea metodei **clone** definită în clasa `Object`. Aceasta creează un nou obiect şi iniţializează toate variabilele sale membre cu valorile obiectului clonat.

```
TipReferinta o1 = new TipReferinta();
TipReferinta o2 = (TipReferinta) o1.clone();
```

Deficienţa acestei metode este că nu realizează duplicarea întregii reţele de obiecte corespunzătoare obiectului clonat. In cazul în care există câmpuri referinţa la alte obiecte, obiectele referite nu vor mai fi clonate la rândul lor.

O metodă `clone` care să realizeze o copie efectivă a unui obiect, împreuna cu copierea tuturor obiectelor referite de câmpurile acelui obiect poate fi implementată prin mecanismul serializării astfel:

```
public Object clone() {
  try {
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    ObjectOutputStream out = new ObjectOutputStream(baos);
    out.writeObject(this);
    out.close();

    byte[] buffer = baos.toByteArray();
    ByteArrayInputStream bais = new ByteArrayInputStream(buffer);
    ObjectInputStream in = new ObjectInputStream(bais);
    Object ret = in.readObject();
    in.close();

    return ret;

  } catch (Exception e) {
```

```
    System.out.println(e);
    return null;
  }
}
```

Capitolul 9

Interfaţa grafică cu utilizatorul

9.1 Introducere

Interfaţa grafică cu utilizatorul (GUI), este un termen cu înţeles larg care se referă la toate tipurile de comunicare *vizuală* între un program şi utilizatorii săi. Aceasta este o particularizare a interfeţei cu utilizatorul (UI), prin care vom întelege conceptul generic de interacţiune dintre program şi utilizatori. Limbajul Java pune la dispoziţie numeroase clase pentru implementarea diverselor funcţionalitati UI, însă ne vom ocupa în continuare de cele care permit realizarea intefeţei grafice cu utilizatorul (GUI).

De la apariţia limbajului Java, bibliotecile de clase care oferă servicii grafice au suferit probabil cele mai mari schimbări în trecerea de la o versiune la alta. Acest lucru se datorează, pe de o parte dificultăţii legate de implementarea noţiunii de portabilitate, pe de altă parte nevoii de a integra mecanismele GUI cu tehnologii apărute şi dezvoltate ulterior, cum ar fi *Java Beans*. In momentul actual, există două modalităţi de a crea o aplicaţie cu interfaţă grafică şi anume:

- **AWT** (Abstract Windowing Toolkit) - este API-ul iniţial pus la dispoziţie începând cu primele versiuni de Java;

- **Swing** - parte dintr-un proiect mai amplu numit **JFC** (Java Foundation Classes) creat în urma colaborării dintre Sun, Netscape şi IBM, Swing se bazează pe modelul AWT, extinzând funcţionalitatea acestuia şi adăugând sau înlocuind componente pentru dezvoltarea aplicaţiilor GUI.

Aşadar, este de preferat ca aplicaţiile Java să fie create folosind tehnologia Swing, aceasta punând la dispoziţie o paletă mult mai largă de facilităţi, însă nu vom renunţa complet la AWT deoarece aici există clase esenţiale, reutilizate în Swing.

In acest capitol vom prezenta clasele de bază şi mecanismul de tratare a evenimentelor din AWT, deoarece va fi simplificat procesul de înţelegere a dezvoltării unei aplicaţii GUI, după care vom face trecerea la Swing.

In principiu, crearea unei aplicaţii grafice presupune următoarele lucruri:

- **Design**
  - Crearea unei suprafeţe de afişare (cum ar fi o fereastră) pe care vor fi aşezate obiectele grafice (componente) care servesc la comunicarea cu utilizatorul (butoane, controale pentru editarea textelor, liste, etc);
  - Crearea şi aşezarea componentelor pe suprafaţa de afişare la poziţiile corespunzătoare;
- **Funcţionalitate**
  - Definirea unor acţiuni care trebuie să se execute în momentul când utilizatorul interacţionează cu obiectele grafice ale aplicaţiei;
  - "Ascultarea" evenimentelor generate de obiecte în momentul interacţiunii cu utilizatorul şi executarea acţiunilor corespunzătoare, aşa cum au fost ele definite.

9.2 Modelul AWT

Pachetul care oferă componente AWT este **java.awt**.
Obiectele grafice sunt derivate din **Component**, cu excepţia meniurilor care descind din clasa **MenuComponent**. Aşadar, prin noţiunea de componentă vom întelege în continuare orice obiect care poate avea o reprezentare grafică şi care poate interactiona cu utilizatorul. Exemple de componente sunt ferestrele, butoanele, listele, bare de defilare, etc. Toate componentele AWT sunt definte de clase proprii ce se gasesc în pachetul `java.awt`, clasa `Component` fiind superclasa abstractă a tuturor acestor clase.

Crearea obiectelor grafice nu realizează automat şi afişarea lor pe ecran. Mai întâi ele trebuie aşezate pe o *suprafata de afişare*, care poate fi o fereastră

sau un applet, şi vor deveni vizibile în momentul în care suprafaţa pe care sunt afişate va fi vizibilă. O astfel de suprafaţă pe care sunt plasate componente se mai numeşte *container* şi reprezintă o instanţă a unei clase derivate din **Container**. Clasa `Container` este o subclasă aparte a lui `Component`, fiind la rândul ei superclasa tuturor suprafetelor de afişare Java.

Aşa cum am văzut, interfaţă grafică serveşte interacţiunii cu utilizatorul. De cele mai multe ori programul trebuie să facă o anumită prelucrare în momentul în care utilizatorul a efectuat o acţiune şi, prin urmare, componentele trebuie să genereze evenimente în funcţie de acţiunea pe care au suferit-o (acţiune transmisă de la tastatură, mouse, etc.). Incepând cu versiunea 1.1 a limbajului Java, evenimentele sunt instanţe ale claselor derivate din **AWTEvent**.
Aşadar, un *eveniment* este produs de o acţiune a utilizatorului asupra unui obiect grafic, deci evenimentele nu trebuie generate de programator. In schimb, într-un program trebuie specificat codul care se execută la apariţia unui eveniment. Tratarea evenimentelor se realizează prin intermediul unor clase de tip *listener* (ascultător, consumator de evenimente), clase care sunt definite în pachetul **java.awt.event**. In Java, orice componentă poate "consuma" evenimentele generate de o altă componentă (vezi "Tratarea evenimentelor").

Să considerăm un mic exemplu, în care creăm o fereastră ce conţine două butoane.

Listing 9.1: O fereastră cu două butoane

```
import java.awt.*;
public class ExempluAWT1 {
  public static void main(String args[]) {

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

    // Setarea modului de dipunere a componentelor
    f.setLayout(new FlowLayout());

    // Crearea celor doua butoane
    Button b1 = new Button("OK");
    Button b2 = new Button("Cancel");

    // Adaugarea butoanelor
    f.add(b1);
```

```
    f.add(b2);
    f.pack();

    // Afisarea fereastrei
    f.show();
  }
}
```

După cum veţi observa la execuţia acestui program, atât butoanele adăugate de noi cât şi butonul de închidere a ferestrei sunt funcţionale, adică pot fi apasate, dar nu realizează nimic. Acest lucru se întâmplă deoarece nu am specificat nicăieri codul care trebuie să se execute la apăsarea acestor butoane.
De asemenea, mai trebuie remarcat că nu am specificat nicăieri dimensiunile ferestrei sau ale butoanelor şi nici pozitiile în acestea să fie plasate. Cu toate acestea ele sunt plasate unul lânga celalalt, fără să se suprapună iar suprafaţa fereastrei este suficient de mare cât să cuprindă ambele obiecte. Aceste "fenomene" sunt provocate de un obiect special de tip `FlowLayout` pe care l-am specificat şi care se ocupă cu gestionarea ferestrei şi cu plasarea componentelor într-o anumită ordine pe suprafaţa ei. Aşadar, modul de aranjare nu este o caracteristică a suprafeţei de afişare ci, fiecare container are asociat un obiect care se ocupă cu dimensionarea şi dispunerea componentelor pe suprafaţa de afişare şi care se numeste *gestionar de poziţionare (layout manager)* (vezi "Gestionarea poziţionării").

9.2.1 Componentele AWT

După cum am spus deja, toate componentele AWT sunt definte de clase proprii ce se gasesc în pachetul `java.awt`, clasa **Component** fiind superclasa abstracta a tuturor acestor clase.

- **Button** - butoane cu eticheta formată dintr-un text pe o singură linie;
- **Canvas** - suprafaţă pentru desenare;
- **Checkbox** - componentă ce poate avea două stări; mai multe obiecte de acest tip pot fi grupate folosind clasa **CheckBoxGroup**;
- **Choice** - liste în care doar elementul selectat este vizibil şi care se deschid la apăsarea lor;

- **Container** - superclasa tuturor suprafeţelor de afişare (vezi "Suprafeţe de afişare");
- **Label** - etichete simple ce pot conţine o singură linie de text needitabil;
- **List** - liste cu selecţie simplă sau multiplă;
- **Scrollbar** - bare de defilare orizontale sau verticale;
- **TextComponent** - superclasa componentelor pentru editarea textului: **TextField** (pe o singură linie) şi **TextArea** (pe mai multe linii).

Mai multe informaţii legate de aceste clase vor fi prezentate în secţiunea "Folosirea componentelor AWT".

Din cauza unor diferenţe esentiale în implementarea meniurilor pe diferite platforme de operare, acestea nu au putut fi integrate ca obiecte de tip `Component`, superclasa care descrie meniuri fiind **MenuComponent** (vezi "Meniuri").

Componentele AWT au peste 100 de metode comune, moştenite din clasa `Component`. Acestea servesc uzual pentru aflarea sau setarea atributelor obiectelor, cum ar fi: dimensiune, poziţie, culoare, font, etc. şi au formatul general **get***Proprietate*, respectiv **set***Proprietate*. Cele mai folosite, grupate pe tipul proprietăţii gestionate sunt:

- **Poziţie**
  `getLocation, getX, getY, getLocationOnScreen`
  `setLocation, setX, setY`
- **Dimensiuni**
  `getSize, getHeight, getWidth`
  `setSize, setHeight, setWidth`
- **Dimensiuni şi poziţie**
  `getBounds`
  `setBounds`
- **Culoare** (text şi fundal)
  `getForeground, getBackground`
  `setForeground, setBackground`

- **Font**
  `getFont`
  `setFont`
- **Vizibilitate**
  `setVisible`
  `isVisible`
- **Interactivitate**
  `setEnabled`
  `isEnabled`

9.2.2 Suprafeţe de afişare (Clasa Container)

Crearea obiectelor grafice nu realizează automat şi afişarea lor pe ecran. Mai întâi ele trebuie aşezate pe o suprafaţă, care poate fi o fereastră sau suprafaţa unui applet, şi vor deveni vizibile în momentul în care suprafaţa respectivă va fi vizibilă. O astfel de suprafaţa pe care sunt plasate componentele se numeşte *suprafaţă de afişare* sau *container* şi reprezintă o instanţa a unei clase derivată din **Container**. O parte din clasele a căror părinte este `Container` este prezentată mai jos:

- **Window** - este superclasa tututor ferestrelor. Din această clasă sunt derivate:
  - **Frame** - ferestre standard;
  - **Dialog** - ferestre de dialog modale sau nemodale;
- **Panel** - o suprafaţă fără reprezentare grafică folosită pentru gruparea altor componente. Din această clasă derivă **Applet**, folosită pentru crearea appleturilor.
- **ScrollPane** - container folosit pentru implementarea automată a derulării pe orizontală sau verticală a unei componente.

Aşadar, un container este folosit pentru a adăuga componente pe suprafaţa lui. Componentele adăugate sunt memorate într-o listă iar poziţiile lor din această listă vor defini ordinea de traversare "front-to-back" a acestora în cadrul containerului. Dacă nu este specificat nici un index la adăugarea unei componente, atunci ea va fi adaugată pe ultima poziţie a listei.

Clasa `Container` conţine metodele comune tututor suprafeţelor de afişare. Dintre cele mai folosite, amintim:

- **add** - permite adăugarea unei componente pe suprafaţa de afişare. O componentă nu poate aparţine decât unui singur container, ceea ce înseamnă că pentru a muta un obiect dintr-un container în altul trebuie sa-l eliminam mai întâi de pe containerul initial.

- **remove** - elimină o componentă de pe container;

- **setLayout** - stabileşte gestionarul de poziţionare al containerului (vezi "Gestionarea poziţionării");

- **getInsets** - determină distanţa rezervată pentru marginile suprafeţei de afişare;

- **validate** - forţează containerul să reaşeze toate componentele sale. Această metodă trebuie apelată explicit atunci când adăugăm sau eliminăm componente pe suprafaţa de afişare după ce aceasta a devenit vizibilă.

Exemplu:

```
Frame f = new Frame("O fereastra");

// Adaugam un buton direct pe fereastra
Button b = new Button("Hello");
f.add(b);

// Adaugam doua componente pe un panel
Label et = new Label("Nume:");
TextField text = new TextField();

Panel panel = new Panel();
panel.add(et);
panel.add(text);

// Adaugam panel-ul pe fereastra
// si, indirect, cele doua componente
f.add(panel);
```

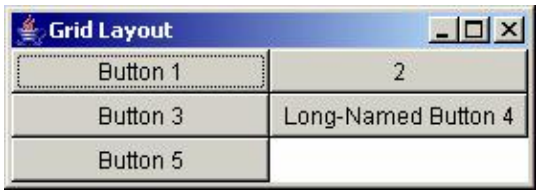
9.3 Gestionarea poziţionării

Să considerăm mai întâi un exemplu de program Java care afişează 5 butoane pe o fereastră:

Listing 9.2: Poziţionarea a 5 butoane

```
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 afişata de acest program va arăta astfel:



Să modificăm acum linia marcata cu '*' ca mai jos, lăsând neschimbat restul programului:

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

Fereastra afişată după această modificare va avea o cu totul altfel de dispunere a componentelor sale:

Motivul pentru care cele două ferestre arată atât de diferit este că folosesc gestionari de poziţionare diferiţi: `GridLayout`, respectiv `FlowLayout`.

Definiţie
Un *gestionar de poziţionare (layout manager)* este un obiect care controlează dimensiunea şi aranjarea (poziţia) componentelor unui container.

Aşadar, modul de aranjare a componentelor pe o suprafaţa de afişare nu este o caracteristică a containerului. Fiecare obiect de tip `Container` (`Applet, Frame, Panel`, etc.) are asociat un obiect care se ocupă cu dispunerea componentelor pe suprafaţa sa şi anume gestionarul său de poziţionare. Toate clasele care instanţiaza obiecte pentru gestionarea poziţionării implementează interfaţă **LayoutManager**.

La instanţierea unui container se creează implicit un gestionar de poziţionare asociat acestuia. De exemplu, pentru o fereastră gestionarul implict este de tip `BorderLayout`, în timp ce pentru un panel este de tip `FlowLayout`.

9.3.1 Folosirea gestionarilor de poziţionare

Aşa cum am văzut, orice container are un gestionar implicit de poziţionare - un obiect care implemenează interfaţa `LayoutManager`, acesta fiindu-i ataşat automat la crearea sa. In cazul în care acesta nu corespunde necesităţilor noastre, el poate fi schimbat cu uşurinţă. Cei mai utilizaţi gestionari din pachetul `java.awt` sunt:

- **FlowLayout**
- **BorderLayout**
- **GridLayout**
- **CardLayout**
- **GridBagLayout**

Pe lângă aceştia, mai există şi cei din modelul Swing care vor fi prezentaţi în capitolul dedicat dezvoltării de aplicaţii GUI folosind Swing.

Ataşarea explicită a unui gestionar de poziţionare la un container se face cu metoda **setLayout** a clasei `Container`. Metoda poate primi ca parametru orice instanţă a unei clase care implementează interfaţă `LayoutManager`. Secvenţa de ataşare a unui gestionar pentru un container, particularizată pentru `FlowLayout`, este:

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

// sau, mai uzual:

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

Programele nu apelează în general metode ale gestionarilor de poziţionare, dar în cazul când avem nevoie de obiectul gestionar îl putem obţine cu metoda **getLayout** din clasa `Container`.

Una din facilităţile cele mai utile oferite de gestionarii de poziţionare este rearanjarea componentele unui container atunci când acesta este redimesionat. Poziţiile şi dimensiunile componentelor nu sunt fixe, ele fiind ajustate automat de către gestionar la fiecare redimensionare astfel încât să ocupe cât mai "estetic" suprafaţa de afişare. Cum sunt determinate însă dimensiunile implicite ale componentelor ?
Fiecare clasă derivată din `Component` poate implementa metodele **getPreferredSize**, **getMinimumSize** şi **getMaximumSize** care să returneze dimensiunea implicită a componentei respective şi limitele în afara cărora componenta nu mai poate fi desenată. Gestionarii de poziţionare vor apela aceste metode pentru a calcula dimensiunea la care vor afişa o componentă.

Sunt însă situaţii când dorim să plasăm componentele la anumite poziţii fixe iar acestea să ramâna acolo chiar dacă redimensionăm containerul. Folosind un gestionar această *poziţionare absolută* a componentelor nu este posibilă şi deci trebuie cumva să renunţăm la gestionarea automată a containerul. Acest lucru se realizează prin trimitera argumentului **null** metodei `setLayout`:

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

Folosind poziţionarea absolută, nu va mai fi însă suficient să adăugam cu metoda `add` componentele în container, ci va trebui să specificăm poziţia şi

dimensiunea lor - acest lucru era făcut automat de gestionarul de poziţionare.

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

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

In general, se recomandă folosirea gestionarilor de poziţionare în toate situaţiile când acest lucru este posibil, deoarece permit programului să aibă aceeaşi "înfatisare" indiferent de platforma şi rezoluţia pe care este rulat. Poziţionarea absolută poate ridica diverse probleme în acest sens.

Să analizam în continuare pe fiecare din gestionarii amintiţi anterior.

9.3.2 Gestionarul FlowLayout

Acest gestionar aşează componentele pe suprafaţa de afişare în flux liniar, mai precis, componentele sunt adăugate una după alta pe linii, în limita spaţiului disponibil. In momentul când o componentă nu mai încape pe linia curentă se trece la următoarea linie, de sus în jos. Adăugarea 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 dreapta şi pe centru. Implicit, componentele sunt centrate pe fiecare linie iar distanţa implicită între componente este de 5 pixeli pe verticală şi 5 pe orizontală.

Este gestionarul implicit al containerelor derivate din clasa `Panel` deci şi al applet-urilor.

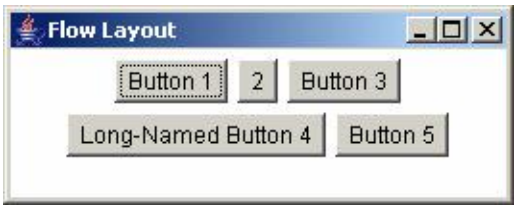
Listing 9.3: Gestionarul `FlowLayout`

```
import java.awt.*;
public class TestFlowLayout {
  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 afişate astfel:

Redimensionând fereastra astfel încât cele cinci butoane să nu mai încapă pe o linie, ultimele dintre ele vor fi trecute pe linia următoare:



9.3.3 Gestionarul BorderLayout

Gestionarul `BorderLayout` împarte suprafaţa de afişare în cinci regiuni, corespunzătoare celor patru puncte cardinale şi centrului. O componentă poate fi plasată în oricare din aceste regiuni, dimeniunea componentei fiind calculata astfel încât să ocupe întreg spaţiul de afişare oferit de regiunea respectivă. Pentru a adăuga 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 dorită (vezi "Gruparea componentelor - clasa Panel").

Aşadar, la adăugarea unei componente pe o suprafaţă gestionată de `BorderLayout`, metoda `add` va mai primi pe lânga referinţa componentei şi zona în care aceasta va fi amplasată, care va fi specificată prin una din constantele clasei: `NORTH, SOUTH, EAST, WEST, CENTER`.

`BorderLayout` este gestionarul implicit pentru toate containerele care descind din clasa `Window`, deci al tuturor tipurilor de ferestre.

Listing 9.4: Gestionarul `BorderLayout`

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

    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 afişate astfel:

La redimensionarea ferestrei se pot observa următoarele lucruri: nordul şi sudul se redimensionează doar pe orizontală, estul şi vestul doar pe verticală, în timp ce centrul se redimensionează atât pe orizontală cât şi pe verticală. Redimensionarea componentelor din fiecare zonă se face astfel încât ele ocupă toată zona containerului din care fac parte.

9.3.4 Gestionarul GridLayout

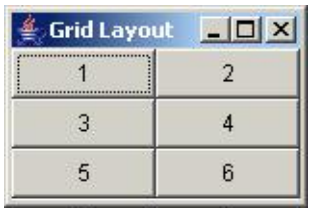
Gestionarul `GridLayout` organizează containerul ca un tabel cu rânduri şi coloane, componentele fiind plasate în celulele tabelului de la stânga la dreapta, începând cu primul rând. Celulele tabelului au dimensiuni egale iar o componentă poate ocupa doar o singură celulă. Numărul de linii şi coloane vor fi specificate în constructorul gestionarului, dar pot fi modificate

şi ulterior prin metodele `setRows`, respectiv `setCols`. Dacă numărul de linii sau coloane este 0 (dar nu ambele în acelaşi timp), atunci componentele vor fi plasate într-o singură coloană sau linie. De asemenea, distanţa între componente pe orizontală şi distanta între rândurile tabelului pot fi specificate în constructor sau stabilite ulterior.

Listing 9.5: Gestionarul `GridLayout`

```
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 şase butoane ale ferestrei vor fi plasate pe trei rânduri şi două coloane, astfel:



Redimensionarea ferestrei va determina redimensionarea tuturor celulelor şi deci a tuturor componentelor, atât pe orizontală cât şi pe verticală.

9.3.5 Gestionarul CardLayout

Gestionarul `CardLayout` tratează componentele adăugate pe suprafaţa sa într-o manieră similară cu cea a dispunerii cărţilor de joc într-un pachet.

Suprafaţa de afişare poate fi asemănată cu pachetul de cărţi iar fiecare componentă este o carte din pachet. La un moment dat, numai o singură componentă este vizibilă ("cea de deasupra").

Clasa dispune de metode prin care să poată fi afişată o anumită componentă din pachet, sau să se poată parcurge secvenţial pachetul, ordinea componentelor fiind internă gestionarului.

Principala utilitate a acestui gestionar este utilizarea mai eficientă a spaţiului disponibil în situaţii în care componentele pot fi grupate în aşa fel încât utilizatorul să interacţioneze la un moment dat doar cu un anumit grup (o carte din pachet), celelalte fiind ascunse.
O clasă Swing care implementează un mecansim similar este `JTabbedPane`.

Listing 9.6: Gestionarul `CardLayout`

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

public class TestCardLayout extends Frame implements
    ActionListener {
  Panel tab;
  public TestCardLayout() {
    super("Test CardLayout");
    Button card1 = new Button("Card 1");
    Button card2 = new Button("Card 2");

    Panel butoane = new Panel();
    butoane.add(card1);
    butoane.add(card2);

    tab = new Panel();
    tab.setLayout(new CardLayout());

    TextField tf = new TextField("Text Field");
    Button btn  = new Button("Button");
    tab.add("Card 1", tf);
    tab.add("Card 2", btn);

    add(butoane, BorderLayout.NORTH);
    add(tab, BorderLayout.CENTER);

    pack();
    show();
```

```
    card1.addActionListener(this);
    card2.addActionListener(this);
  }

  public void actionPerformed(ActionEvent e) {
    CardLayout gestionar = (CardLayout) tab.getLayout();
    gestionar.show(tab, e.getActionCommand());
  }

  public static void main(String args[]) {
    TestCardLayout f = new TestCardLayout();
    f.show();
  }
}
```

Prima "carte" este vizibilă

A doua "carte" este vizibilă

9.3.6 Gestionarul GridBagLayout

Este cel mai complex şi flexibil gestionar de poziţionare din Java. La fel ca în cazul gestionarului `GridLayout`, suprafaţa de afişare este considerată ca fiind un tabel însă, spre deosebire de acesta, numărul de linii şi de coloane sunt determinate automat, în funcţie de componentele amplasate pe suprafaţa de afişare. De asemenea, în funcţie de componentele gestionate, dimensiunile celulelor pot fi diferite cu singurele restricţii ca pe aceeaşi linie să aibă aceeaşi înalţime, iar pe coloană aibă aceeaşi lăţime. Spre deosebire de `GridLayout`, o componentă poate ocupa mai multe celule adiacente, chiar de dimensiuni diferite, zona ocupată fiind referită prin "regiunea de afişare" a componentei respective.

Pentru a specifica modul de afişare a unei componente, acesteia îi este asociat un obiect de tip **GridBagConstraints**, în care se specifică diferite proprietăţi ale componentei referitoare la regiunea să de afişare şi la modul în care va fi plasată în această regiune. Legătura dintre o componentă şi un obiect `GridBagConstraints` se realizează prin metoda **setConstraints**:

```
GridBagLayout gridBag = new GridBagLayout();
```

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

Aşadar, înainte de a adăuga o componentă pe suprafaţa unui container care are un gestionar de tip `GridBagLayout`, va trebui să specificăm anumiţi parametri (constrângeri) referitori la cum va fi plasată componenta respectivă. Aceste constrângeri vor fi specificate prin intermediul unui obiect de tip `GridBagConstraints`, care poate fi refolosit pentru mai multe componente care au aceleaşi constrângeri de afişare:

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

Cele mai utilizate tipuri de constrângeri pot fi specificate prin intermediul următoarelor variabile din clasa `GridBagConstraints`:

- `gridx, gridy` - celula ce reprezintă colţul stânga sus al componentei;
- `gridwidth, gridheight` - numărul de celule pe linie şi coloană pe care va fi afişată componenta;
- `fill` - folosită pentru a specifica dacă o componentă va ocupa întreg spaţiul pe care îl are destinat; valorile posibile sunt `HORIZONTAL`, `VERTICAL`, `BOTH, NONE`;
- `insets` - distanţele dintre componentă şi marginile suprafeţei sale de afişare;
- `anchor` - folosită atunci când componenta este mai mică decât suprafaţa sa de afişare pentru a forţa o anumită dispunere a sa: nord, sud, est, vest, etc.
- `weigthx, weighty` - folosite pentru distribuţia spaţiului liber; uzual au valoarea 1;

Ca exemplu, să realizăm o fereastră ca în figura de mai jos. Pentru a simplifica codul, a fost creată o metodă responsabilă cu setarea valorilor `gridx, gridy, gridwidth, gridheight` şi adăugarea unei componente cu restricţiile stabilite pe fereastră.

Listing 9.7: Gestionarul `GridBagLayout`

```
import java.awt.*;

public class TestGridBagLayout {
  static Frame f;
  static GridBagLayout gridBag;
  static GridBagConstraints gbc;

  static void adauga(Component comp,
        int x, int y, int w, int h) {
    gbc.gridx = x;
    gbc.gridy = y;
    gbc.gridwidth = w;
    gbc.gridheight = h;

    gridBag.setConstraints(comp, gbc);
    f.add(comp);
  }

  public static void main(String args[]) {

    f = new Frame("Test GridBagLayout");
    gridBag = new GridBagLayout();

    gbc = new GridBagConstraints();
    gbc.weightx = 1.0;
    gbc.weighty = 1.0;
```

```
    gbc.insets = new Insets(5, 5, 5, 5);

    f.setLayout(gridBag);

    Label mesaj = new Label("Evidenta persoane", Label.CENTER
       );
    mesaj.setFont(new Font("Arial", Font.BOLD, 24));
    mesaj.setBackground(Color.yellow);
    gbc.fill = GridBagConstraints.BOTH;
    adauga(mesaj, 0, 0, 4, 2);

    Label etNume = new Label("Nume:");
    gbc.fill = GridBagConstraints.NONE;
    gbc.anchor = GridBagConstraints.EAST;
    adauga(etNume, 0, 2, 1, 1);

    Label etSalariu = new Label("Salariu:");
    adauga(etSalariu, 0, 3, 1, 1);

    TextField nume = new TextField("", 30);
    gbc.fill = GridBagConstraints.HORIZONTAL;
    gbc.anchor = GridBagConstraints.CENTER;
    adauga(nume, 1, 2, 2, 1);

    TextField salariu = new TextField("", 30);
    adauga(salariu, 1, 3, 2, 1);

    Button adaugare = new Button("Adaugare");
    gbc.fill = GridBagConstraints.NONE;
    adauga(adaugare, 3, 2, 1, 2);

    Button salvare = new Button("Salvare");
    gbc.fill = GridBagConstraints.HORIZONTAL;
    adauga(salvare, 1, 4, 1, 1);

    Button iesire = new Button("Iesire");
    adauga(iesire, 2, 4, 1, 1);

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

9.3.7 Gruparea componentelor (Clasa Panel)

Plasarea componentelor direct pe suprafaţa de afişare poate deveni incomodă în cazul în care avem multe obiecte grafice. Din acest motiv, se recomandă gruparea componentelor înrudite ca funcţii astfel încât să putem fi siguri că, indiferent de gestionarul de poziţionare al suprafeţei de afişare, ele se vor găsi împreună. Gruparea componentelor se face în **panel-uri**.

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

Aşadar, o aranjare eficientă a componentelor unei ferestre înseamnă:

- gruparea componentelor "înfraţite" (care nu trebuie să fie desparţite de gestionarul de poziţionare al ferestrei) în panel-uri;
- aranjarea componentelor unui panel, prin specificarea unui gestionar de poziţionare corespunzător;
- aranjarea panel-urilor pe suprafaţa ferestrei, prin specificarea gestionarului de poziţionare al ferestrei.

Listing 9.8: Gruparea componentelor

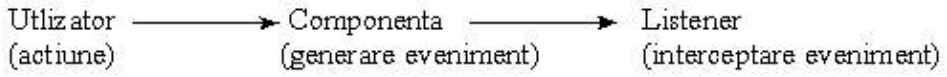
```
import java.awt.*;
public class TestPanel {
  public static void main(String args[]) {
    Frame f = new Frame("Test Panel");

    Panel intro = new Panel();
    intro.setLayout(new GridLayout(1, 3));
    intro.add(new Label("Text:"));
    intro.add(new TextField("", 20));
    intro.add(new Button("Adaugare"));

    Panel lista = new Panel();
    lista.setLayout(new FlowLayout());
    lista.add(new List(10));
    lista.add(new Button("Stergere"));
```

```
    Panel control = new Panel();
    control.add(new Button("Salvare"));
    control.add(new Button("Iesire"));

    f.add(intro, BorderLayout.NORTH);
    f.add(lista, BorderLayout.CENTER);
    f.add(control, BorderLayout.SOUTH);

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

9.4 Tratarea evenimentelor

Un *eveniment* este produs de o acţiune a utilizatorului asupra unei componente grafice şi reprezintă mecanismul prin care utilizatorul comunică efectiv cu programul. Exemple de evenimente sunt: apăsarea unui buton, modificarea textului într-un control de editare, închiderea sau redimensionarea unei ferestre, etc. Componentele care generează anumite evenimente se mai numesc şi *surse de evenimente*.

Interceptarea evenimentelor generate de componentele unui program se realizează prin intermediul unor clase de tip **listener** (ascultător, consumator de evenimente). In Java, orice obiect poate "consuma" evenimentele generate de o anumită componentă grafică.



Aşadar, pentru a scrie cod care să se execute în momentul în care utilizatorul interactionează cu o componentă grafică trebuie să facem următoarele lucruri:

- să scriem o clasă de tip listener care să "asculte" evenimentele produse de acea componentă şi în cadrul acestei clase să implementăm metode specifice pentru tratarea lor;

- să comunicăm componentei sursă că respectiva clasa îi "ascultă" evenimentele pe care le generează, cu alte cuvinte să înregistrăm acea clasă drept "consumator" al evenimentelor produse de componenta respectivă.

Evenimentele sunt, ca orice altceva în Java, obiecte. Clasele care descriu aceste obiecte se împart în mai multe tipuri în funcţie de componenta care le generează, mai precis în funcţie de acţiunea utilizatorului asupra acesteia. Pentru fiecare tip de eveniment există o clasă care instanţiază obiecte de acel tip. De exemplu, evenimentul generat de acţionarea unui buton este descris de clasa `ActionEvent`, cel generat de modificarea unui text de clasa `TextEvent`, etc. Toate aceste clase sunt derivate din superclasa **AWTEvent**, lista lor completă fiind prezentată ulterior.

O clasă consumatoare de evenimente (listener) poate fi orice clasă care specifica în declaraţia sa că doreşte să asculte evenimente de un anumit tip. Acest lucru se realizează prin implementarea unei interfeţe specifice fiecărui tip de eveniment. Astfel, pentru ascultarea evenimentelor de tip `ActionEvent` clasa respectivă trebuie să implementeze interfaţa `ActionListener`, pentru `TextEvent` interfaţă care trebuie implementata este `TextListener`, etc. Toate aceste interfeţe sunt derivate din **EventListener**.

Fiecare interfaţă defineşte una sau mai multe metode care vor fi apelate automat la apariţia unui eveniment:

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

Intrucât o clasă poate implementa oricâte interfeţe, ea va putea să asculte evenimente de mai multe tipuri:

```
class Ascultator implements ActionListener, TextListener {
  public void actionPerformed(ActionEvent e) { ... }
  public void textValueChanged(TextEvent e) { ... }
}
```

Vom vedea în continuare metodele fiecărei interfeţe pentru a şti ce trebuie să implementeze o clasă consumatoare de evenimente.

Aşa cum am spus mai devreme, pentru ca evenimentele unei componente să fie interceptate de către o instanţă a unei clase ascultător, această clasă trebuie înregistrata în lista ascultătorilor componentei respective. Am spus lista, deoarece evenimentele unei componente pot fi ascultate de oricâte clase, cu condiţia ca acestea să fie înregistrate la componenta respectivă. Inregistrarea unei clase în lista ascultătorilor unei componente se face cu metode din clasa `Component` de tipul **add***TipEveniment***Listener**, iar eliminarea ei din această listă cu **remove***TipEveniment***Listener**.

Sumarizând, tratarea evenimentelor în Java se desfăşoară astfel:

- Componentele generează evenimente când ceva "interesant" se întâmplă;
- Sursele evenimentelor permit oricărei clase să "asculte" evenimentele sale prin metode de tip `addXXXListener`, unde XXX este un tip de eveniment;
- O clasă care ascultă evenimente trebuie să implementeze interfeţe specifice fiecărui tip de eveniment - acestea descriu metode ce vor fi apelate automat la apariţia evenimentelor.

9.4.1 Exemplu de tratare a evenimentelor

Inainte de a detalia aspectele prezentate mai sus, să considerăm un exemplu de tratare a evenimentelor. Vom crea o fereastră care să conţină două butoane cu numele "OK", repectiv "Cancel". La apăsarea fiecărui buton vom scrie pe bara de titlu a ferestrei mesajul "Ati apasat butonul ...".

Listing 9.9: Ascultarea evenimentelor a două butoane

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

```
class Fereastra extends Frame {
  public Fereastra(String titlu) {
    super(titlu);
    setLayout(new FlowLayout());
    setSize(200, 100);
    Button b1 = new Button("OK");
    Button b2 = new Button("Cancel");
    add(b1);
    add(b2);

    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) {
   f.setTitle("Ati apasat " + e.getActionCommand());
  }
}

public class TestEvent1 {
  public static void main(String args[]) {
    Fereastra f = new Fereastra("Test Event");
    f.show();
  }
}
```

Nu este obligatoriu să definim clase speciale pentru ascultarea evenimentelor. In exemplul de mai sus am definit clasa `Ascultator` pentru a intercepta evenimentele produse de cele două butoane şi din acest motiv a trebuit să trimitem ca parametru constructorului clasei o referinţa la fereastra noastră. Mai simplu ar fi fost să folosim chiar clasa `Fereastra` pentru a trata evenimentele produse de componentele sale. Vom modifica puţin şi

aplicaţia pentru a pune în evidenţa o altă modalitate de a determina componenta generatoare a unui eveniment - metoda **getSource**.

Listing 9.10: Tratarea evenimentelor în ferestră

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

class Fereastra extends Frame implements ActionListener {
  Button ok = new Button("OK");
  Button exit = new Button("Exit");
  int n=0;

  public Fereastra(String titlu) {
    super(titlu);
    setLayout(new FlowLayout());
    setSize(200, 100);
    add(ok);
    add(exit);

    ok.addActionListener(this);
    exit.addActionListener(this);
    // Ambele butoane sunt ascultate in clasa Fereastra
    // deci ascultatorul este instanta curenta: this
  }

  // Metoda interfetei ActionListener
  public void actionPerformed(ActionEvent e) {

    if (e.getSource() == exit)
        System.exit(0); // Terminam aplicatia

    if (e.getSource() == ok) {
        n ++;
        this.setTitle("Ati apasat OK de " + n + " ori");
    }
  }
}

public class TestEvent2 {
  public static void main(String args[]) {
    Fereastra f = new Fereastra("Test Event");
    f.show();
  }
}
```

Aşadar, orice clasă poate asculta evenimente de orice tip cu condiţia să implementeze interfeţele specifice acelor tipuri de evenimente.

9.4.2 Tipuri de evenimente

Evenimentele se împart în două categorii: *de nivel jos* şi *semantice*.

Evenimentele de nivel jos reprezintă o interacţiune de nivel jos cum ar fi o apăsare de tastă, mişcarea mouse-ului, sau o operaţie asupra unei ferestre. In tabelul de mai jos sunt enumerate clasele ce descriu aceste evenimente şi operaţiunile efectuate (asupra unei componente) care le generează:

`ComponentEvent`	Ascundere, deplasare, redimensionare, afişare
`ContainerEvent`	Adăugare pe container, eliminare
`FocusEvent`	Obţinere, pierdere foucs
`KeyEvent`	Apăsare, eliberare taste, tastare
`MouseEvent`	Operaţiuni cu mouse-ul: click, drag, etc.
`WindowEvent`	Operaţiuni asupra ferestrelor: minimizare, maximizare,etc.

O anumită acţiune a utilizatorului poate genera mai multe evenimente. De exemplu, tastarea literei 'A' va genera trei evenimente: unul pentru apăsare, unul pentru eliberare şi unul pentru tastare. In funcţie de necesităţile aplicaţie putem scrie cod pentru tratarea fiecărui eveniment în parte.

Evenimentele semantice reprezintă interacţiunea cu o componentă GUI: apăsarea unui buton, selectarea unui articol dintr-o listă, etc. Clasele care descriu aceste tipuri de evenimente sunt:

`ActionEvent`	Acţionare
`AdjustmentEvent`	Ajustarea unei valori
`ItemEvent`	Schimbarea stării
`TextEvent`	Schimbarea textului

Următorul tabel prezintă componentele AWT şi tipurile de evenimente generate, prezentate sub forma interfeţelor corespunzătoare. Evident, evenimentele generate de o superclasă, cum ar fi `Component`, se vor regăsi şi pentru toate subclasele sale.

`Component`	`ComponentListener`
	`FocusListener`
	`KeyListener`
	`MouseListener`
	`MouseMotionListener`

`Container`	`ContainerListener`
`Window`	`WindowListener`

`Button`
`List`
`MenuItem`
`TextField`
	`ActionListener`

`Choice`
`Checkbox`
`List`
`CheckboxMenuItem`
	`ItemListener`

`Scrollbar`	`AdjustmentListener`
`TextField` `TextArea`	`TextListener`

Observaţi că deşi există o singură clasă `MouseEvent`, există două interfeţe asociate `MouseListener` şi `MouseMotionListener`. Acest lucru a fost făcut deoarece evenimentele legate de deplasarea mouse-ului sunt generate foarte frecvent şi recepţionarea lor poate avea un impact negativ asupra vitezei de execuţie, în situaţia când tratarea acestora nu ne interesează şi dorim să tratăm doar evenimente de tip click, de exemplu.

Orice clasă care tratează evenimente trebuie să implementeze obligatoriu metodele interfeţelor corespunzătoare. Tabelul de mai jos prezintă, pentru fiecare interfaţă, metodele puse la dispozitie şi care trebuie implementate de către clasa ascultător.

Interfaţă	**Metode**
`ActionListener`	`actionPerformed(ActionEvent e)`
`AdjustmentListener`	`adjustmentValueChanged(AdjustmentEvent e)`

`ComponentListener`	`componentHidden(ComponentEvent e)`
	`componentMoved(ComponentEvent e)`
	`componentResized(ComponentEvent e)`
	`componentShown(ComponentEvent e)`

`ContainerListener`	`componentAdded(ContainerEvent e)` `componentRemoved(ContainerEvent e)`
`FocusListener`	`focusGained(FocusEvent e)` `focusLost(FocusEvent e)`
`ItemListener`	`itemStateChanged(ItemEvent e)`

`KeyListener`	`keyPressed(KeyEvent e)`
	`keyReleased(KeyEvent e)`
	`keyTyped(KeyEvent e)`

`MouseListener`	`mouseClicked(MouseEvent e)`
	`mouseEntered(MouseEvent e)`
	`mouseExited(MouseEvent e)`
	`mousePressed(MouseEvent e)`
	`mouseReleased(MouseEvent e)`

`MouseMotionListener`	`mouseDragged(MouseEvent e)` `mouseMoved(MouseEvent e)`
`TextListener`	`textValueChanged(TextEvent e)`

`WindowListener`	`windowActivated(WindowEvent e)`
	`windowClosed(WindowEvent e)`
	`windowClosing(WindowEvent e)`
	`windowDeactivated(WindowEvent e)`
	`windowDeiconified(WindowEvent e)`
	`windowIconified(WindowEvent e)`
	`windowOpened(WindowEvent e)`

In cazul în care un obiect listener tratează evenimente de acelaşi tip provocate de componente diferite, este necesar să putem afla, în cadrul uneia din metodele de mai sus, care este sursa evenimentului pe care îl tratăm pentru a putea reacţiona în consecinţă. Toate tipurile de evenimente moştenesc metoda **getSource** care returnează obiectul responsabil cu generarea evenimentului. In cazul în care dorim să diferenţiem doar tipul componentei sursă,

putem folosi operatorul **instanceof**.

```
public void actionPerformed(ActionEvent e) {
  Object sursa = e.getSource();
  if (sursa instanceof Button) {
    // A fost apasat un buton
    Button btn = (Button) sursa;
    if (btn == ok) {
      // A fost apasat butonul 'ok'
    }
    ...
  }
  if (sursa instanceof TextField) {
    // S-a apasat Enter dupa editarea textului
    TextField tf = (TextField) sursa;
    if (tf == nume) {
      // A fost editata componenta 'nume'
    }
    ...
  }
}
```

Pe lângă `getSource`, obiectele ce descriu evenimente pot pune la dispoziţie şi alte metode specifice care permit aflarea de informaţii legate de evenimentul generat. De exemplu, `ActionEvent` conţine metoda `getActionCommand` care, implicit, returnează eticheta butonului care a fost apăsat. Astfel de particularităţi vor fi prezentate mai detaliat în secţiunile dedicate fiecărei componente în parte.

9.4.3 Folosirea adaptorilor şi a claselor anonime

Am vazut că o clasă care tratează evenimente de un anumit tip trebuie să implementeze interfaţa corespunzătoare acelui tip. Aceasta înseamnă că trebuie să implementeze obligatoriu toate metodele definite de acea interfaţă, chiar dacă nu specifică nici un cod pentru unele dintre ele. Sunt însă situaţii când acest lucru poate deveni supărator, mai ales atunci când nu ne interesează decât o singura metodă a interfeţei.

Un exemplu sugestiv este următorul: o fereastră care nu are specificat cod pentru tratarea evenimentelor sale nu poate fi închisă cu butonul standard

marcat cu 'x' din colţul dreapta sus şi nici cu combinaţia de taste Alt+F4. Pentru a realiza acest lucru trebuie interceptat evenimentul de închidere a ferestrei în metoda `windoClosing` şi apelată metoda `dispose` de închidere a ferestrei, sau `System.exit` pentru terminarea programului, în cazul când este vorba de fereastra principală a aplicaţiei. Aceasta înseamnă că trebuie să implementăm interfaţa `WindowListener` care are nu mai puţin de **şapte** metode.

Listing 9.11: Implementarea interfeţei `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) {
    // Terminare program
    System.exit(0);
  }
  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("Test WindowListener");
    f.show();
  }
}
```

Observaţi că trebuie să implementăm toate metodele interfeţei, chiar dacă nu scriem nici un cod pentru unele dintre ele. Singura metodă care ne interesează este `windowClosing`, în care specificăm ce trebuie făcut atunci când utilizatorul doreste să închidă fereastra. Pentru a evita scrierea inutilă a

acestor metode, există o serie de clase care implementează interfeţele de tip "listener" fără a specifica nici un cod pentru metodele lor. Aceste clase se numesc **adaptori**.

Un *adaptor* este o clasă abstractă care implementează o anumită interfaţă fără a specifica cod nici unei metode a interfeţei.

Scopul unei astfel de clase este ca la crearea unui "ascultător" de evenimente, în loc să implementă o anumită interfaţă şi implicit toate metodele sale, să extindem adaptorul corespunzător interfeţei respective (dacă are!) şi să supradefinim doar metodele care ne interesează (cele în care vrem să scriem o anumită secvenţă de cod).

De exemplu, adaptorul interfeţei `WindowListener` este `WindowAdapter` iar folosirea acestuia este dată în exemplul de mai jos:

Listing 9.12: 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);
  }
}

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

Avantajul clar al acestei modalităţi de tratare a evenimentelor este reducerea codului programului, acesta devenind mult mai lizibil. Insă există şi două dezavantaje majore. După cum aţi observat faţăde exemplul anterior,

clasa `Fereastra` nu poate extinde `WindowAdapter` deoarece ea extinde deja clasa `Frame` şi din acest motiv am construit o nouă clasă numită `Ascultator`. Vom vedea însă că acest dezavantaj poate fi eliminat prin folosirea unei clase anonime.
Un alt dezavantaj este că orice greşeală de sintaxă în declararea unei metode a interfeţei nu va produce o eroare de compilare dar nici nu va supradefini metoda interfeţei ci, pur şi simplu, va crea o metodă 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 daţi toţi adaptorii interfeţelor de tip "listener" - se obervă că o interfaţă `XXXListener` are un adaptor de tipul `XXXAdapter`. Interfeţele care nu au un adaptor sunt cele care definesc o singură metodă şi prin urmare crearea unei clase adaptor nu îsi are rostul.

Interfaţa	**Adaptor**
`ActionListener`	nu are
`AdjustemnrListener`	nu are
`ComponentListener`	`ComponentAdapter`
`ContainerListener`	`ContainerAdapter`
`FocusListener`	`FocusAdapter`
`ItemListener`	nu are
`KeyListener`	`KeyAdapter`
`MouseListener`	`MouseAdapter`
`MouseMotionListener`	`MouseMotionAdapter`
`TextListener`	nu are
`WindowListener`	`WindowAdapter`

Ştim că o clasă internă este o clasă declarată în cadrul altei clase, iar clasele anonime sunt clase interne folosite pentru instanţierea unui singur obiect de un anumit tip. Un exemplu tipic de folosire a lor este instanţierea

adaptorilor direct în corpul unei clase care conţine componente ale căror evenimente trebuie tratate.

Listing 9.13: Folosirea adaptorilor şi a claselor anonime

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

class Fereastra extends Frame {
  public Fereastra(String titlu) {
    super(titlu);
    setSize(400, 400);

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

    final Label label = new Label("", Label.CENTER);
    label.setBackground(Color.yellow);
    add(label, BorderLayout.NORTH);


    this.addMouseListener(new MouseAdapter() {
      public void mouseClicked(MouseEvent e) {
        // Desenam un cerc la fiecare click de mouse
        label.setText("Click... ");
        Graphics g = Fereastra.this.getGraphics();
        g.setColor(Color.blue);
        int raza = (int)(Math.random() * 50);
        g.fillOval(e.getX(), e.getY(), raza, raza);
      }
    });

    this.addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseMoved(MouseEvent e) {
        // Desenam un punct la coordonatele mouse-ului
        Graphics g = Fereastra.this.getGraphics();
        g.drawOval(e.getX(), e.getY(), 1, 1);
      }
    });
```

```
    this.addKeyListener(new KeyAdapter() {
      public void keyTyped(KeyEvent e) {
        // Afisam caracterul tastat
        label.setText("Ati tastat: " + e.getKeyChar() + "");
      }
    });
  }
}

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

9.5 Folosirea ferestrelor

După cum am văzut suprafeţele de afişare ale componentelor sunt extensii ale clasei `Container`. O categorie aparte a acestor containere o reprezintă ferestrele. Acestea sunt descrise de clase derivate din **Window**, cele mai utilizate fiind **Frame** şi **Dialog**.

O aplicaţie Java cu intefaţă grafică va fi formată din una sau mai multe ferestre, una dintre ele fiind numită *fereastra principală*.

9.5.1 Clasa Window

Clasa `Window` este rar utilizată în mod direct deoarece permite doar crearea unor ferestre care nu au chenar şi nici bară de meniuri. Este utilă atunci când dorim să afişam ferestre care nu interacţionează cu utilizatorul ci doar oferă anumite informaţii.

Metodele mai importante ale clasei `Window`, care sunt de altfel moştenite de toate subclasele sale, sunt date de mai jos:

- `show` - face vizibilă fereastra. Implicit, o fereastră nou creată nu este vizibilă;
- `hide` - face fereastra invizibilă fără a o distruge însă; pentru a redeveni vizibila se poate apela metoda `show`;

- `isShowing` - testează dacă fereastra este vizibilă sau nu;
- `dispose` - închide) fereastra şi şi eliberează toate resursele acesteia;
- `pack` - redimensionează automat fereastra la o suprafaţa optimă care să cuprindă toate componentele sale; trebuie apelată în general după adăugarea tuturor componentelor pe suprafaţa ferestrei.
- `getFocusOwner` - returnează componenta ferestrei care are focus-ul (dacă fereastra este activă).

9.5.2 Clasa Frame

Este derivată a clasei `Window` şi este folosită pentru crearea de ferestre independente şi funcţionale, eventual continând o bară de meniuri. Orice aplicaţie cu interfaţă grafică contţine cel puţin o fereastră, cea mai importantă fiind numită şi *fereastra principală*.

Constructorii uzuali ai clasei Frame permit crearea unei ferestre cu sau fără titlu, iniţial invizibilă. Pentru ca o fereastră să devină vizibilă se va apela metoda `show` definită în superclasa `Window`.

```
import java.awt.*;
public class TestFrame {
  public static void main(String args[]) {
    Frame f = new Frame("Titlul ferestrei");
    f.show();
  }
}
```

Crearea ferestrelor prin instanţierea directă a obiectelor de tip `Frame` este mai puţin folosită. 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);
    ...
  }
```

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

Gestionarul de poziţionare implicit al clasei `Frame` este `BorderLayout`. Din acest motiv, în momentul în care fereastra este creată dar nici o componentă grafică nu este adăugată, suprafaţa de afişare a feretrei va fi determinată automota de gestionarul de pozitţionare şi va oferi doar spaţiul necesar afişării barei ferestrei şi grupului de butoane pentru minimizare, maximizare şi închidere. Acelaşi efect îl vom obţine dacă o redimenionam şi apelăm apoi metoda `pack` care determină dimeniunea suprafeţei de afişare în funcţie de componentele adăugate.

Se observă de asemenea că butonul de închidere a ferestrei nu este funcţional. Tratarea evenimentelor ferestrei se face prin implementarea interfeţei `WindowListener` sau, mai uzual, prin folosirea unui adaptor de tip `WindowAdapter`.

Structura generală a unei ferestre este descrisă de clasa `Fereastra` din exemplul de mai jos:

Listing 9.14: Structura generală a unei ferestre

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

class Fereastra extends Frame implements ActionListener {

  // Constructorul
  public Fereastra(String titlu) {
    super(titlu);

    // Tratam evenimentul de inchidere a ferestrei
    this.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        dispose(); // inchidem fereastra
        // sau terminam aplicatia
        System.exit(0);
      }
    });
```

```
    // Eventual, schimbam gestionarul de pozitionare
    setLayout(new FlowLayout());

    // Adaugam componentele pe suprafata ferestrei
    Button exit = new Button("Exit");
    add(exit);

    // Facem inregistrarea claselor listener
    exit.addActionListener(this);

    // Stabilim dimensiunile
    pack(); // implicit

    //sau explicit
    // setSize(200, 200);
  }

  // Implementam metodele interfetelor de tip listener
  public void actionPerformed(ActionEvent e) {
    System.exit(0);
  }
}

public class TestFrame {
  public static void main(String args[]) {
    // Cream fereastra
    Fereastra f = new Fereastra("O fereastra");

    // O facem vizibila
    f.show();
  }
}
```

Pe lângă metodele moştenite din clasa `Window`, există o serie de metode specifice clasei Frame. Dintre cele mai folosite amintim:

- `getFrames` - metodă statică ce returnează lista tuturor ferestrelor deschise ale unei aplicaţii;
- `setIconImage` - setează iconiţa ferestrei;
- `setMenuBar` - setează bara de meniuri a ferestrei (vezi "Folosirea meniurilor");
- `setTitle` - setează titlul ferestrei;

- `setResizable` - stabileşte dacă fereastra poate fi redimenionată de utilizator;

9.5.3 Clasa Dialog

Toate interfeţele grafice oferă un tip special de ferestre destinate preluării unor informaţii sau a unor date de la utilizator. Acestea se numesc *ferestre de dialog* sau *casete de dialog* şi sunt implementate prin intermediul clasei **Dialog**, subclasă directă a clasei `Window`.

Diferenţa majoră dintre ferestrele de dialog şi ferestrele de tip `Frame` constă în faptul că o fereastră de dialog este dependentă de o altă fereastra (normală sau tot fereastră de dialog), numită şi *fereastra părinte*. Cu alte cuvinte, ferestrele de dialog nu au o existenţă de sine stătătoare. Când fereastra părinte este distrusă sunt distruse şi ferestrele sale de dialog, când este minimizată ferestrele sale de dialog sunt făcute invizibile iar când este restaurată acestea sunt aduse la starea în care se găseau în momentul minimizării ferestrei părinte.

Ferestrele de dialog pot fi de două tipuri:

- **modale**: care blochează accesul la fereastra parinte în momentul deschiderii lor, cum ar fi ferestrele de introducere a unor date, de alegere a unui fişier, de selectare a unei opţiuni, mesaje de avertizare, etc;

- **nemodale**: care nu blochează fluxul de intrare către fereastra părinte - de exemplu, dialogul de căutare a unui cuvânt într-un fişier, etc.

Implicit, o fereastră de dialog este nemodală şi invizibilă, însă există constructori care să specifice şi aceşti parametri. 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)
```

Parametrul "părinte" reprezintă referinţa la fereastra părinte, "titlu" reprezintă titlul ferestrei iar prin argumentul "modală" specificăm dacă fereastra de dialog creată va fi modală (`true`) sau nemodală (`false` - valoarea implicită).

Crearea unei ferestre de dialog este relativ simpla şi se realizează prin derivarea clasei `Dialog`. Comunicarea dintre fereastra de dialog şi fereastra sa părinte, pentru ca aceasta din urmă să poată folosi datele introduse (sau opţiunea specificata) în caseta de dialog, se poate realiza folosind una din următoarele abordări generale:

- obiectul care reprezintă dialogul poate să trateze evenimentele generate de componentele de pe suprafaţa să şi să seteze valorile unor variabile accesibile ale ferestrei părinte în momentul în care dialogul este încheiat;
- obiectul care creează dialogul (fereastra părinte) să se înregistreze ca ascultător al evenimentelor de la butoanele care determină încheierea dialogului, iar fereastra de dialog să ofere metode publice prin care datele introduse să fie preluate din exterior;

Să creăm, de exemplu, o fereastră de dialog modală pentru introducerea unui şir de caractere. Fereastra principală a aplicatiei va fi părintele casetei de dialog, va primi şirul de caractere introdus şi îsi va modifica titlul ca fiind acesta. Deschiderea ferestrei de dialog se va face la apăsarea unui buton al ferestrei principale numit "Schimba titlul". Cele două ferestre vor arăta ca în imaginile de mai jos:

Fereastra principală

Fereastra de dialog

Listing 9.15: Folosirea unei ferestre de dialog

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

// Fereastra principala
class FerPrinc extends Frame implements ActionListener{
```

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

    setLayout(new FlowLayout());
    setSize(300, 80);
    Button b = new Button("Schimba titlul");
    add(b);
    b.addActionListener(this);
  }

  public void actionPerformed(ActionEvent e) {
    FerDialog d = new FerDialog(this, "Dati titlul", true);
    String titlu = d.raspuns;
    if (titlu == null)
      return;
    setTitle(titlu);
  }
}

// Fereastra de dialog
class FerDialog extends Dialog implements ActionListener {
  public String raspuns = null;
  private TextField text;
  private Button ok, cancel;

  public FerDialog(Frame parinte, String titlu, boolean
     modala) {
    super(parinte, titlu, modala);
    this.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        raspuns = null;
        dispose();
      }
    });

    text = new TextField("", 30);
    add(text, BorderLayout.CENTER);

    Panel panel = new Panel();
    ok = new Button("OK");
```

```
    cancel = new Button("Cancel");
    panel.add(ok);
    panel.add(cancel);

    add(panel, BorderLayout.SOUTH);
    pack();

    text.addActionListener(this);
    ok.addActionListener(this);
    cancel.addActionListener(this);

    show();
  }

  public void actionPerformed(ActionEvent e) {
    Object sursa = e.getSource();
    if (sursa == ok || sursa == text)
      raspuns = text.getText();
    else
      raspuns = null;
    dispose();
  }
}

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

9.5.4 Clasa FileDialog

Pachetul `java.awt` pune la dispozitie şi un tip de fereastră de dialog folosită pentru selectarea unui nume de fişier în vederea încărcării sau salvării unui fişier: clasa `FileDialog`, derivată din `Dialog`. Instanţele acestei clase au un comportament comun dialogurilor de acest tip de pe majoritatea platformelor de lucru, dar forma în care vor fi afişate este specifică platformei pe care rulează aplicaţia.

Constructorii clasei sunt:

```
FileDialog(Frame parinte)
```

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

Parametrul "părinte" reprezintă referinţa ferestrei părinte, "titlu" reprezintă titlul ferestrei iar prin argumentul "mod" specificăm dacă încărcăm sau salvăm un fişier; valorile pe care le poate lua acest argument sunt:

- `FileDialog.LOAD` - pentru încărcare, respectiv
- `FileDialog.SAVE` - pentru salvare.

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

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

La crearea unui obiect `FileDialog` acesta nu este implicit vizibil. Dacă afişarea sa se face cu `show`, caseta de dialog va fi modală. Dacă afişarea se face cu `setVisible(true)`, atunci va fi nemodală. După selectarea unui fişier ea va fi facută automat invizibilă.

Pe lângă metodele moştenite de la superclasa `Dialog` clasa `FileDialog` mai conţine metode pentru obţinerea numelui fişierului sau directorului selectat `getFile, getDirectory`, pentru stabilirea unui criteriu de filtrare `setFilenameFilter`, etc.

Să considerăm un exemplu în care vom alege, prin intermediul unui obiect `FileDialog`, un fişier cu extensia "java". Directorul iniţial este directorul curent, iar numele implicit este `TestFileDialog.java`. Numele fişierului ales va fi afişat la consolă.

Listing 9.16: Folosirea unei ferestre de dialog

```
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);
      }
    });

    Button b = new Button("Alege fisier");
    add(b, BorderLayout.CENTER);
    pack();

    b.addActionListener(this);
  }

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

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

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

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

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

Clasa `FileDialog` este folosită mai rar deoarece în Swing există clasa `JFileChooser` care oferă mai multe facilităţi şi prin urmare va constitui prima opţiune într-o aplicaţie cu intefaţă grafică.

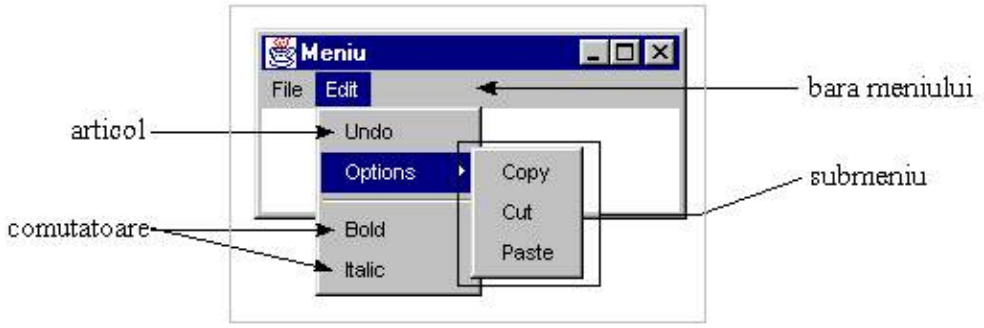
9.6 Folosirea meniurilor

Spre deosebire de celelalte obiecte grafice care derivă din clasa `Component`, componentele unui meniu reprezintă instanţe ale unor clase derivate din superclasa abstractă **MenuComponent**. Această excepţie este facută deoarece unele platforme grafice limitează capabilităţile unui meniu.

Meniurile pot fi grupate în două categorii:

- **Meniuri fixe** (vizibile permanent): sunt grupate într-o bară de meniuri ce conţine câte un meniu pentru fiecare intrare a sa. La rândul lor, aceste meniuri conţin articole ce pot fi selectate, comutatoare sau alte meniuri (submeniuri). O fereastră poate avea un singur meniu fix.
- **Meniuri de context** (popup): sunt meniuri invizbile asociate unei ferestre şi care se activează uzual prin apăsarea butonului drept al mouse-ului. O altă diferenţă faţă de meniurile fixe constă în faptul că meniurile de context nu sunt grupate într-o bară de meniuri.

In figura de mai jos este pusă în evidenţă alcătuirea unui meniu fix:



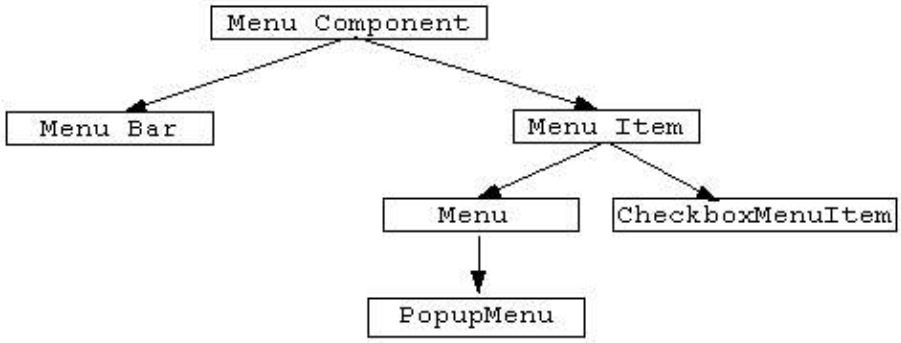
Exemplul de mai sus conţine o bară de meniuri formată din două meniuri principale *File* şi *Edit*. Meniul *Edit* conţine la rândul lui alt meniu (submeniu) *Options*, articolul *Undo* şi două comutatoare *Bold* şi *Italic*. Prin abuz de limbaj, vom referi uneori bara de meniuri a unei ferestre ca fiind meniul ferestrei.

In modelul AWT obiectele care reprezintă bare de meniuri sunt reprezentate ca instanţe al clasei **MenuBar**. Un obiect de tip `MenuBar` conţine obiecte de tip **Menu**, care sunt de fapt meniurile derulante propriu-zise. La rândul lor, acestea pot conţine obiecte de tip **MenuItem, CheckBoxMenuItem**, dar şi alte obiecte de tip `Menu` (submeniuri).

Pentru a putea conţine un meniu, o componentă trebuie să implementeze interfaţa **MenuContainer**. Cel mai adesea, meniurile sunt ataşate ferestrelor, mai precis obiectelor de tip `Frame`, acestea implementând interfaţă `MenuContainer`. Ataşarea unei bare de meniuri la o fereastră se face prin metoda **addMenuBar** a clasei `Frame`.

9.6.1 Ierarhia claselor ce descriu meniuri

Să vedem în continuare care este ierarhia claselor folosite în lucrul cu meniuri şi să analizăm pe rând aceste clase:



Clasa **MenuComponent** este o clasa abstractă din care sunt extinse toate celelalte clase folosite la crearea de meniuri, fiind similară celeilalte superclase abstracte `Component`. `MenuComponent` conţine metode cu caracter general, dintre care amintim `getName, setName, getFont, setFont`, cu sintaxa şi semnificaţiile uzuale.

Clasa **MenuBar** permite crearea barelor de meniuri asociate unei ferestre cadru de tip `Frame`, adaptând conceptul de bară de meniuri la platforma curentă de lucru. După cum am mai spus, pentru a lega bara de meniuri la o anumită fereastra trebuie apelată metoda `setMenuBar` din clasa `Frame`.

```
// Crearea barei de meniuri
MenuBar mb = new MenuBar();
```

```
// Adaugarea meniurilor derulante la bara de meniuri
...

// Atasarea barei de meniuri la o fereastra
Frame f = new Frame("Fereastra cu meniu");
f.addMenuBar(mb);
```

Orice articol al unui meniu trebuie să fie o instanţa a clasei **MenuItem**. Obiectele acestei clase descriu aşadar opţiunile individuale ale meniurilor derulante, cum sunt "Open", "Close", "Exit", etc. O instanţă a clasei `MenuItem` reprezintă de fapt un buton sau un comutator, cu o anumită etichetă care va apărea în meniu, însoţită eventual de un accelerator (obiect de tip `MenuShortcut`) ce reprezintă combinaţia de taste cu care articolul poate fi apelat rapid (vezi "Acceleratori").

Clasa **Menu** permite crearea unui meniu derulant într-o bară de meniuri. Opţional, un meniu poate fi declarat ca fiind *tear-off*, ceea ce înseamnă că poate fi deschis şi deplasat cu mouse-ul (dragged) într-o altă poziţie decât cea originală ("rupt" din poziţia sa). Acest mecanism este dependent de platformă şi poate fi ignorat pe unele dintre ele. Fiecare meniu are o etichetă, care este de fapt numele său ce va fi afişat pe bara de meniuri. Articolele dintr-un meniu trebuie să aparţină clasei `MenuItem`, ceea ce înseamnă că pot fi instanţe ale uneia din clasele `MenuItem, Menu sau CheckboxMenuItem`.

Clasa **CheckboxMenuItem** implementează într-un meniu articole de tip comutator - care au două stări logice (validat/nevalidat), acţionarea articolului determinând trecerea sa dintr-o stare în alta. La validarea unui comutator în dreptul etichetei sale va fi afişat un simbol grafic care indică acest lucru; la invalidarea sa, simbolul grafic respectiv va dispărea. Clasa `CheckboxMenuItem` are aceeaşi funcţionalitate cu cea a casetelor de validare de tip `Checkbox`, ambele implementând interfaţa `ItemSelectable`.

Să vedem în continuare cum ar arăta un program care construieşte un meniu ca în figura prezentată anterior:

Listing 9.17: Crearea unui meniu

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

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

    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.setSize(200, 100);
    f.show();
  }
}
```

9.6.2 Tratarea evenimentelor generate de meniuri

La alegerea unei opţiuni dintr-un meniu se generează fie un eveniment de tip **ActionEvent** dacă articolul respectiv este de tip `MenuItem`, fie **ItemEvent** pentru comutatoarele `CheckboxMenuItem`. Aşadar, pentru a activa opţiunile unui meniu trebuie implementate interfaţele **ActionListener** sau/şi **ItemListener** în cadrul obiectelor care trebuie să specifice codul ce va fi executat la alegerea unei opţiuni şi implementate metodele `actionPerformed`, respectiv `itemStatChanged`. Fiecărui meniu îi putem asocia un obiect receptor diferit, ceea ce uşurează munca în cazul în care ierarhia de meniuri este complexă. Pentru a realiza legătura între obiectul meniu şi obiectul de tip listener trebuie să adaugăm receptorul în lista de ascultători a meniului respectiv, întocmai ca pe orice componentă, folosind metodele `addActionListener`, respectiv `addItemListener`.

Aşadar, tratarea evenimentelor generate de obiecte de tip `MenuItem` este identică cu tratarea butoanelor, ceea ce face posibil ca unui buton de pe suprafaţa de afişare să îi corespundă o opţiune dintr-un meniu, ambele cu acelaşi nume, tratarea evenimentului corespunzător apăsarii butonului, sau alegerii opţiunii, făcându-se o singură dată într-o clasă care este înregistrată ca receptor atât la buton cât şi la meniu.

Obiectele de tip `CheckboxMenuItem` tip se găsesc într-o categorie comună cu `List, Choice, CheckBox`, toate implementând interfaţa `ItemSelectable` şi deci tratarea lor va fi făcută la fel. Tipul de operatie selectare / deselectare este codificat în evenimentul generat de câmpurile statice `ItemEvent.SELECTED` şi `ItemEvent.DESELECTED`.

Listing 9.18: Tratarea evenimentelor unui meniu

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

public class TestMenuEvent extends Frame
     implements ActionListener, ItemListener {

  public TestMenuEvent(String titlu) {
    super(titlu);

    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);
    setMenuBar(mb);

    Button btnExit = new Button("Exit");
    add(btnExit, BorderLayout.SOUTH);
    setSize(300, 200);
    show();

    test.addActionListener(this);
    check.addItemListener(this);
    btnExit.addActionListener(this);
  }

  public void actionPerformed(ActionEvent e) {
    // Valabila si pentru meniu si pentru buton
    String command = e.getActionCommand();
    if (command.equals("Exit"))
      System.exit(0);
  }

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

  public static void main(String args[]) {
    TestMenuEvent f = new TestMenuEvent("Tratare evenimente
       meniuri");
    f.show();
  }
}
```

9.6.3 Meniuri de context (popup)

Au fost introduse începând cu AWT 1.1 şi sunt implementate prin intermediul clasei **PopupMenu**, subclasă directă a clasei `Menu`. Sunt meniuri invizibile care sunt activate uzual prin apăsarea butonului drept al mouse-ului, fiind

afişate la poziţia la care se găsea mouse-ul în momentul apăsării butonului său drept. Metodele de adăugare a articolelor unui meniu de context sunt moştenite întocmai de la meniurile fixe.

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

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

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

şi este de obicei rezultatul apăsarii unui buton al mouse-ului, pentru a avea acces rapid la meniu. Argumentul "origine" reprezintă componenta faţă de originile căreia se va calcula poziţia de afişare a meniului popup. De obicei, reprezintă instanţa ferestrei în care se va afişa meniul. Deoarece interacţiunea cu mouse-ul este dependentă de platforma de lucru, există o metodă care determină dacă un eveniment de tip `MouseEvent` poate fi responsabil cu deschiderea unui meniu de context. Aceasta este `isPopupTrigger` şi este definită în clasa `MouseEvent`. Poziţionarea şi afişarea meniului este însă responsabilitatea programatorului.

Meniurile de context nu se adaugă la un alt meniu (bară sau sub-meniu) ci se ataşează la o componentă (de obicei la o fereastră) prin metoda **add** a acesteia. In cazul când avem mai multe meniuri popup pe care vrem să le folosim într-o fereastră, trebuie să le definim pe toate şi, la un moment dat, vom adăuga ferestrei meniul corespunzător după care îl vom face vizibil. După închiderea acestuia, vom "rupe" legătura între fereastră şi meniu prin instrucţiunea **remove**:

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

In exemplul de mai jos, vom crea un meniu de contex pe care îl vom activa la apăsarea butonului drept al mouse-ului pe suprafaţa ferestrei principale. Tratarea evenimentelor generate de un meniu popup se realizează identic ca pentru meniurile fixe.

Listing 9.19: Folosirea unui meniu de context (popup)

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

class Fereastra extends Frame implements ActionListener{
  // Definim meniul popup al ferestrei
  private PopupMenu popup;
  // Pozitia meniului va fi relativa la fereastra
  private Component origin;
  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 mousePressed(MouseEvent e) {
        if (e.isPopupTrigger())
          popup.show(origin, e.getX(), e.getY());
      }
      public void mouseReleased(MouseEvent e) {
        if (e.isPopupTrigger())
          popup.show(origin, e.getX(), e.getY());
      }
    });
    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();
    if (command.equals("Exit"))
      System.exit(0);
```

```
  }
}

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

9.6.4 Acceleratori (Clasa MenuShortcut)

Pentru articolele unui menu este posibilă specificarea unor combinaţii de taste numite *acceleratori* (shortcuts) care să permită accesarea directă, prin intermediul tastaturii, a opţiunilor dintr-un meniu. Astfel, oricărui obiect de tip `MenuItem` îi poate fi asociat un obiect de tip accelerator, definit prin intermediul clasei **MenuShortcut**. Singurele combinaţii de taste care pot juca rolul acceleratorilor sunt: `Ctrl + Tasta` sau `Ctrl + Shift + Tasta`. Atribuirea unui accelerator la un articol al unui meniu poate fi realizată prin constructorul obiectelor de tip `MenuItem` în forma:
`MenuItem(String eticheta, MenuShortcut accelerator)`, ca în exemplele de mai jos:

```
// Ctrl+O
new MenuItem("Open", new MenuShortcut(KeyEvent.VK_O));

// Ctrl+P
new MenuItem("Print", new MenuShortcut('p'));

// Ctrl+Shift+P
new MenuItem("Preview", new MenuShortcut('p'), true);
```

9.7 Folosirea componentelor AWT

In continuare vor fi date exemple de folosire ale componentelor AWT, în care să fie puse în evidentţă cât mai multe din particularităţile acestora, precum şi modul de tratare a evenimentelor generate.

9.7.1 Clasa Label

Un obiect de tip `Label` (etichetă) reprezintă o componentă pentru plasarea unui text pe o suprafaţa de afişare. O etichetă este formată dintr-o singură linie de text static ce nu poate fi modificat de către utilizator, dar poate fi modificat din program.

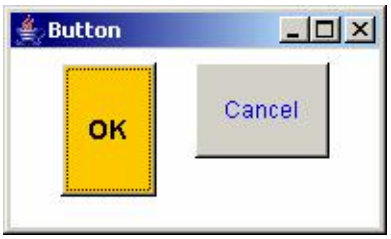
Listing 9.20: Folosirea clasei `Label`

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

    Label nord, sud, est, vest, centru;

    nord = new Label("Nord", Label.CENTER);
    nord.setForeground(Color.blue);

    sud = new Label("Sud", Label.CENTER);
    sud.setForeground(Color.red);

    vest = new Label("Vest", Label.LEFT);
    vest.setFont(new Font("Dialog", Font.ITALIC, 14));

    est = new Label("Est", Label.RIGHT);
    est.setFont(new Font("Dialog", Font.ITALIC, 14));

    centru = new Label("Centru", Label.CENTER);
    centru.setBackground(Color.yellow);
    centru.setFont(new Font("Arial", Font.BOLD, 20));

    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();
  }
}
```

9.7.2 Clasa Button

Un obiect de tip `Button` reprezintă o componentă pentru plasarea unui buton etichetat pe o suprafaţa de afişare. Textul etichetei este format dintr-o singură linie.



Listing 9.21: Folosirea clasei `Button`

```
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);
      }
    });

    setLayout(null);
    setSize(200, 120);

    Button b1 = new Button("OK");
    b1.setBounds(30, 30, 50, 70);
    b1.setFont(new Font("Arial", Font.BOLD, 14));
```

```
    b1.setBackground(Color.orange);
    add(b1);

    Button b2 = new Button("Cancel");
    b2.setBounds(100, 30, 70, 50);
    b2.setForeground(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);
    if (command.equals("OK"))
      setTitle("Confirmare!");
    else
    if (command.equals("Cancel"))
      setTitle("Anulare!");
  }
}

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

9.7.3 Clasa Checkbox

Un obiect de tip `Checkbox` (comutator) reprezintă o componentă care se poate găsi în două stări: "selectată" sau "neselectată" (on/off). Acţiunea utilizatorului asupra unui comutator îl trece pe acesta în starea complementară celei în care se găsea. Este folosit pentru a prelua o anumită opţiune de la utilizator.

Listing 9.22: Folosirea clasei `Checkbox`

```
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);
      }
    });

    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);
```

```
    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.show();
  }
}
```

9.7.4 Clasa CheckboxGroup

Un obiect de tip `CheckboxGroup` defineşte un grup de comutatoare din care doar unul poate fi selectat. Uzual, aceste componente se mai numesc *butoane radio*. Această clasă nu este derivată din `Component`, oferind doar o modalitate de grupare a componentelor de tip `Checkbox`.

Listing 9.23: Folosirea clasei `CheckboxGroup`

```
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);
      }
    });

    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("HBO", cbg, false);
    cbx2 = new Checkbox("Discovery", cbg, false);
    cbx3 = new Checkbox("MTV", cbg, false);

    add(label1);
    add(label2);
    add(cbx1);
```

```
    add(cbx2);
    add(cbx3);

    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.show();
  }
}
```

9.7.5 Clasa Choice

Un obiect de tip `Choice` defineşte o listă de opţiuni din care utilizatorul poate selecta una singură. La un moment dat, din întreaga listă doar o singură opţiune este vizibilă, cea selectată în momentul curent. O componentă `Choice` este însoţită de un buton etichetat cu o sageată verticală la apăsarea căruia este afişată întreaga sa listă de elemente, pentru ca utilizatorul să poată selecta o anumită opţiune.

Listing 9.24: Folosirea clasei `Choice`

```
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);
      }
    });

    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);

    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.show();
  }
}
```

9.7.6 Clasa List

Un obiect de tip `List` defineşte o listă de opţiuni care poate fi setată astfel încât utilizatorul să poată selecta o singură opţiune sau mai multe. Toate elementele listei sunt vizibile în limita dimensiunilor grafice ale componentei.

Listing 9.25: Folosirea clasei `List`

```
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);
      }
    });

    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);

    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.show();
  }
}
```

9.7.7 Clasa ScrollBar

Un obiect de tip `Scrollbar` defineşte o bară de defilare pe verticală sau orizontală. Este utilă pentru punerea la dispoziţia utilizatorului a unei modalităţi sugestive de a alege o anumită valoare dintr-un interval.

Listing 9.26: Folosirea clasei `ScrollBar`

```
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);
      }
    });

    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);

    setSize(200, 80);

    scroll.addAdjustmentListener(this);
  }

  // Metoda interfetei AdjustmentListener
```

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

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

9.7.8 Clasa ScrollPane

Un obiect de tip `ScrollPane` permite ataşarea unor bare de defilare (orizontală şsi/sau verticală) oricărei componente grafice. Acest lucru este util pentru acele componente care nu au implementată funcţionalitatea de defilare automată, cum ar fi listele (obiecte din clasa `List`).

Listing 9.27: Folosirea clasei `ScrollPane`

```
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);
```

```
      }
    });

    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, BorderLayout.CENTER);

    setSize(200, 200);
  }

}

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

9.7.9 Clasa TextField

Un obiect de tip `TextField` defineşte un control de editare a textului pe o singură linie. Este util pentru interogarea utilizatorului asupra unor valori.

Listing 9.28: Folosirea clasei `TextField`

```
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="Duke", PWD="java" ;

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

    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!", Label.
       CENTER);
    add(p1);
    add(p2);
    add(acces);

    setSize(350, 100);

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

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

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

9.7.10 Clasa TextArea

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

Listing 9.29: Folosirea clasei `TextArea`

```
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 salvare;

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

    setBackground(Color.lightGray);
    text = new TextArea("", 30, 10, TextArea.
       SCROLLBARS_VERTICAL_ONLY);
    nume = new TextField("", 12);
    salvare = new Button("Salveaza text");
    salvare.setEnabled(false);

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

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

    setSize(300, 200);

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

  // Metoda interfetei TextListener
  public void textValueChanged(TextEvent e) {
    if (text.getText().length() == 0 ||
        nume.getText().length() == 0)
      salvare.setEnabled(false);
    else
      salvare.setEnabled(true);
```

```
  }

  // Metoda interfetei ActionListener
  public void actionPerformed(ActionEvent e) {
    String continut = text.getText();
    try {
      PrintWriter out = new PrintWriter(
        new FileWriter(nume.getText()));
      out.print(continut);
      out.close();
      text.requestFocus();
    } catch(IOException ex) {
      ex.printStackTrace();
    }
  }
}

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

Capitolul 10

Desenarea

10.1 Conceptul de desenare

Un program Java care are interfaţă grafică cu utilizatorul trebuie să deseneze pe ecran toate componentele sale care au o reprezentare vizuală. Această desenare include componentele standard folosite în aplicaţie precum şi cele definite de către programator. Desenarea componentelor se face automat şi este un proces care se execută în următoarele situaţii:

- la afişarea pentru prima dată a unei componente;
- la operaţii de minimizare, maximizare, redimensionare a suprafeţei de afişare;
- ca răspuns al unei solicitări explicite a programului.

Metodele care controlează procesul de desenare se găsesc în clasa `Component` şi sunt următoarele:

- **void paint(Graphics g)** - Desenează o componentă. Este o metodă supradefinită de fiecare componentă în parte pentru a furniza reprezentarea sa grafică specifică. Metoda este apelată de fiecare dată când conţinutul componentei trebuie desenat sau redesenat şi nu va fi apelată explicit.
- **void update(Graphics g)** - Actualizează starea grafică a unei componente. Acţiunea acestei metode se realizează în trei paşi:
    1. şterge componenta prin supradesenarea ei cu culoarea fundalului;

2. stabileşte culoarea (foreground) a componentei;
3. apelează metoda `paint` pentru a redesena componenta.

- **void repaint()** - Execută explicit un apel al metodei `update` pentru a actualiza reprezentarea grafică a unei componente.

După cum se observă, singurul argument al metodelor `paint` şi `update` este un obiect de tip **Graphics**. Acesta reprezintă *contextul grafic* în care se execută desenarea componentelor (vezi "Contextul grafic de desenare - clasa Graphics").

Atenţie

Toate desenele care trebuie să apară pe o suprafaţa de desenare se realizează în metoda `paint` a unei componente, în general apelată automat sau explicit cu metoda `repaint` ori de câte ori componenta respectivă trebuie redesenată. Există posibilitatea de a desena şi în afara metodei `paint`, însă aceste desene se vor pierde la prima operaţie de minimizare, maximizare, redimensionare a suprafeţei de afişare.

10.1.1 Metoda paint

După cum am spus, toate desenele care trebuie să apară pe o suprafaţă de afişare se realizează în metoda `paint` a unei componente. Metoda `paint` este definită în superclasa `Component` însă nu are nici o implementare şi, din acest motiv, orice obiect grafic care doreşte să se deseneze trebuie să o supradefinească pentru a-şi crea propria sa reprezentare. Componentele standard AWT au deja supradefinită această metodă deci nu trebuie să ne preocupe desenarea lor, însă putem modifica reprezentarea lor grafică prin crearea unei subclase şi supradefinirea metodei `paint`, având însă grijă să apelăm şi metoda superclasei care se ocupă cu desenarea efectivă a componentei.

In exemplul de mai jos, redefinim metoda `paint` pentru un obiect de tip `Frame`, pentru a crea o clasă ce instanţiază ferestre pentru o aplicaţie demonstrativă (în colţul stânga sus este afişat textul "Aplicatie DEMO").

Listing 10.1: Supradefinirea metodei `paint`

```
import java.awt.*;
class Fereastra extends Frame {
  public Fereastra(String titlu) {
    super(titlu);
    setSize(200, 100);
  }

  public void paint(Graphics g) {
    // Apelam metoda paint a clasei Frame
    super.paint(g);
    g.setFont(new Font("Arial", Font.BOLD, 11));
    g.setColor(Color.red);
    g.drawString("Aplicatie DEMO", 5, 35);
  }
}

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

Observaţi că la orice redimensionare a ferestrei textul "Aplicatie DEMO" va fi redesenat. Dacă desenarea acestui text ar fi fost facută oriunde în altă parte decât în metoda `paint`, la prima redimensionare a ferestrei acesta s-ar pierde.

Aşadar, desenarea în Java trebuie să se facă doar în cadrul metodelor `paint` ale componentelor grafice.

10.1.2 Suprafeţe de desenare - clasa Canvas

In afara posibilităţii de a utiliza componente grafice standard, Java oferă şi posibilitatea controlului la nivel de punct (pixel) pe dispozitivul grafic, respectiv desenarea a diferite forme grafice direct pe suprafaţa unei componente. Deşi este posibil, în general nu se desenează la nivel de pixel direct pe suprafaţa ferestrelor sau a altor containere, ci vor fi folosite clase dedicate acestui scop.

In AWT a fost definit un tip special de componentă numită **Canvas** (pânză de pictor), al cărei scop este de a fi extinsă pentru a implementa

obiecte grafice cu o anumită înfăţişare. Aşadar, `Canvas` este o clasă generică din care se derivează subclase pentru crearea suprafeţelor de desenare (planşe). Planşele nu pot conţine alte componente grafice, ele fiind utilizate doar ca suprafeţe de desenat sau ca fundal pentru animaţie. Desenarea pe o planşa se face prin supradefinirea metodei `paint` a acesteia.

Concret, o *planşă* este o suprafaţă dreptunghiulară de culoare albă, pe care se poate desena. Dimensiunile sale implicite sunt 0 şi, din acest motiv, este recomandat ca o planşa să redefinească metoda `getPreferredSize`, eventual şi `getMinimumSize`, `getMaximumSize`, deoarece acestea vor fi apelate de către gestionarii de poziţionare.

Etapele uzuale care trebuie parcurse pentru crearea unui desen, sau mai bine zis a unei componente cu o anumită înfăţişare, sunt:

- crearea unei planşe de desenare, adică o subclasă a lui `Canvas`;
- redefinirea metodei `paint` din clasa respectivă;
- redefinirea metodelor `getPreferredSize`, eventual `getMinimumSize, getMaximumSize`;
- adăugarea planşei pe un container cu metoda `add`.
- tratarea evenimentelor de tip `FocusEvent, KeyEvent, MouseEvent, ComponentEvent`, dacă este cazul.

Definirea generică a unei planşe are următorul format:

```
class Plansa extends Canvas implements ...Listener {

  //Eventual, unul sau mai multi constructori
  public Plansa() {
    ...
  }
  // Metode de desenare a componentei
  public void paint(Graphics g) {
  ...
  }
  // Metodele folosite de gestionarii de pozitionare
  public Dimension getPreferredSize() {
    // Dimensiunea implicita a plansei
```

```
    return ...;
  }
  public Dimension getMinimumSize() {
    return ...
  }
  public Dimension getMaximumSize() {
    return ...
  }
  // Implementarea metodelor interfetelor de tip Listener
  ...
}
```

Să definim o planşă pe care desenăm un pătrat şi cercul său circumscris, colorate diferite. La fiecare click de mouse, vom interschimba cele două culori între ele.

Listing 10.2: Folosirea clasei `Canvas`

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

class Plansa extends Canvas {
  Dimension dim = new Dimension(100, 100);
  private Color color[] = {Color.red, Color.blue};
  private int index = 0;

  public Plansa() {
    this.addMouseListener(new MouseAdapter() {
      public void mouseClicked(MouseEvent e) {
        index = 1 - index;
        repaint();
      }
    });
  }

  public void paint(Graphics g) {
    g.setColor(color[index]);
    g.drawRect(0, 0, dim.width, dim.height);
    g.setColor(color[1 - index]);
    g.fillOval(0, 0, dim.width, dim.height);
  }

  public Dimension getPreferredSize() {
```

```
    return dim;
  }
}

class Fereastra extends Frame {
  public Fereastra(String titlu) {
    super(titlu);
    setSize(200, 200);
    add(new Plansa(), BorderLayout.CENTER);
  }
}

public class TestCanvas {
  public static void main(String args[]) {
    new Fereastra("Test Canvas").show();
  }
}
```

10.2 Contextul grafic de desenare

Inainte ca utilizatorul să poată desena, el trebuie să obţină un context grafic de desenare pentru suprafaţa căreia îi aparţine regiunea pe care se va desena. Un *context grafic* este, de fapt, un obiect prin intermediul căruia putem controla procesul de desenare a unui obiect. In general, desenarea se poate face:

- pe o porţiune de ecran,
- la imprimantă sau
- într-o zonă virtuală de memorie.

Un context grafic este specificat prin intermediul unui obiect de tip **Graphics** primit ca parametru în metodele `paint` şi `update`. In funcţie de dispozitivul fizic pe care se face afişarea (ecran, imprimantă, plotter, etc) metodele de desenare au implementări interne diferite, transparente utilizatorului.

Clasa `Graphics` pune la dispoziţie metode pentru:

- primitive grafice: desenarea de figuri geometrice, texte şi imagini
- stabilirea proprietăţilor contextului grafic, adică stabilirea:

- culorii şi fontului curente cu care se face desenarea,
- originii coordonatelor suprafeţei de desenare,
- suprafeţei în care sunt vizibile componentelor desenate,
- modului de desenare.

10.2.1 Proprietăţile contextului grafic

La orice tip de desenare parametrii legaţi de culoare, font, etc. vor fi specificaţi pentru contextul grafic în care se face desenarea şi nu vor fi trimişi ca argumente metodelor respective de desenare. In continuare, enumerăm aceste proprietăţi şi metodele asociate lor din clasa `Graphics`.

Proprietate	**Metode**
Culoarea de desenare	`Color getColor()` `void setColor(Color c)`
Fontul de scriere a textelor	`Font getFont()` `void setFont(Font f)`
Originea coordonatelor	`translate(int x, int y)`
Zona de decupare (zona în care sunt vizibile desenele)	`Shape getClip()` `void setClip(Shape s)`
Modul de desenare	`void setXorMode(Color c)` `void setPaintMode(Color c)`

10.2.2 Primitive grafice

Prin *primitive grafice* ne vom referi în continuare la metodele clasei `Graphics`, care permit desenarea de figuri geometrice şi texte.

Desenarea textelor se face cu uzual cu metoda `drawString` care primeşte ca argumente un şir şi colţul din stânga-jos al textului. Textul va fi desenat cu fontul şi culoarea curente ale contextului grafic.

```
// Desenam la coordonatele x=10, y=20;
drawString("Hello", 10, 20);
```

Desenarea figurilor geometrice se realizează cu următoarele metode:

Figură geometrică	**Metode**
Linie	`drawLine` `drawPolyline`

Dreptunghi simplu	`drawRect`
	`fillRect`
	`clearRect`

Dreptunghi cu chenar "ridicat" sau "adâncit"	`draw3DRect` `fill3DRect`
Dreptunghi cu colţuri retunjite	`drawRoundRect` `fillRoundRect`
Poligon	`drawPolygon` `fillPolygon`
Oval (Elipsă	`drawOval` `fillOval`
Arc circular sau eliptic	`drawArc` `fillArc`

Metodele care încep cu "fill" vor desena figuri geometrice care au interiorul colorat, adică "umplut" cu culoarea curentă a contextului de desenare, în timp ce metodele care încep cu "draw" vor desena doar conturul figurii respective.

10.3 Folosirea fonturilor

După cum vazut, pentru a scrie un text pe ecran avem două posibilităţi. Prima dintre acestea este să folosim o componentă orientată-text, cum ar fi `Label`, iar a doua să apelăm la metodele clasei `Graphics` de desenare a textelor, cum ar fi `drawString`. Indiferent de modalitatea aleasă, putem specifica prin intermediul fonturilor cum să arate textul respectiv, acest lucru realizându-se prin metoda **setFont** fie din clasa `Component`, fie din `Graphics`.

Cei mai importanţi parametri ce caracterizează un font sunt:

- Numele fontului: Helvetica Bold, Arial Bold Italic, etc.
- Familia din care face parte fontul: Helvetica, Arial, etc.
- Dimensiunea fontului: înălţimea sa;
- Stilul fontului: **îngroşat (bold)**, *înclinat (italic)*;

- Metrica fontului.

Clasele care oferă suport pentru lucrul cu fonturi sunt **Font** şi **FontMetrics**, în continuare fiind prezentate modalităţile de lucru cu acestea.

10.3.1 Clasa Font

Un obiect de tip `Font` încapsulează informaţii despre toţi parametrii unui font, mai puţin despre metrica acestuia. Constructorul uzual al clasei este cel care primeşte ca argument numele fontului, dimensiunea şi stilul acestuia:

```
Font(String name, int style, int size)
```

Stilul unui font este specificat prin intermediul constantelor: `Font.PLAIN`, `Font.BOLD`, `Font.ITALIC` iar dimensiunea printr-un întreg, ca în exemplele de mai jos:

```
new Font("Dialog", Font.PLAIN, 12);
new Font("Arial", Font.ITALIC, 14);
new Font("Courier", Font.BOLD, 10);
```

Folosirea unui obiect de tip `Font` se realizează uzual astfel:

```
// Pentru componente etichetate
  Label label = new Label("Un text");
  label.setFont(new Font("Dialog", Font.PLAIN, 12));

// In metoda paint(Graphics g)
  g.setFont(new Font("Courier", Font.BOLD, 10));
  g.drawString("Alt text", 10, 20);
```

O platformă de lucru are instalate, la un moment dat, o serie întreagă de fonturi care sunt disponibile pentru scrierea textelor. Lista acestor fonturi se poate obţine astfel:

```
Font[] fonturi = GraphicsEnvironment.
  getLocalGraphicsEnvironment().getAllFonts();
```

Exemplul urmator afişează lista tuturor fonturilor disponibile pe platforma curentă de lucru. Textul fiecărui nume de font va fi scris cu fontul său corespunzător.

Listing 10.3: Lucrul cu fonturi

```
import java.awt.*;

class Fonturi extends Canvas {
  private Font[] fonturi;
  Dimension canvasSize = new Dimension(400, 400);

  public Fonturi() {
    fonturi = GraphicsEnvironment.
      getLocalGraphicsEnvironment().getAllFonts();
    canvasSize.height = (1 + fonturi.length) * 20;
  }

  public void paint(Graphics g) {
    String nume;
    for(int i=0; i < fonturi.length; i++) {
      nume = fonturi[i].getFontName();
      g.setFont(new Font(nume, Font.PLAIN, 14));
      g.drawString(i + ". " + nume, 20, (i + 1) * 20);
    }
  }

  public Dimension getPreferredSize() {
    return canvasSize;
  }
}

class Fereastra extends Frame {
  public Fereastra(String titlu) {
    super(titlu);
    ScrollPane sp = new ScrollPane();
    sp.setSize(400, 400);
    sp.add(new Fonturi());
    add(sp, BorderLayout.CENTER);
    pack();
  }
}

public class TestAllFonts {
  public static void main(String args[]) {
    new Fereastra("All fonts").show();
  }
}
```

10.3.2 Clasa FontMetrics

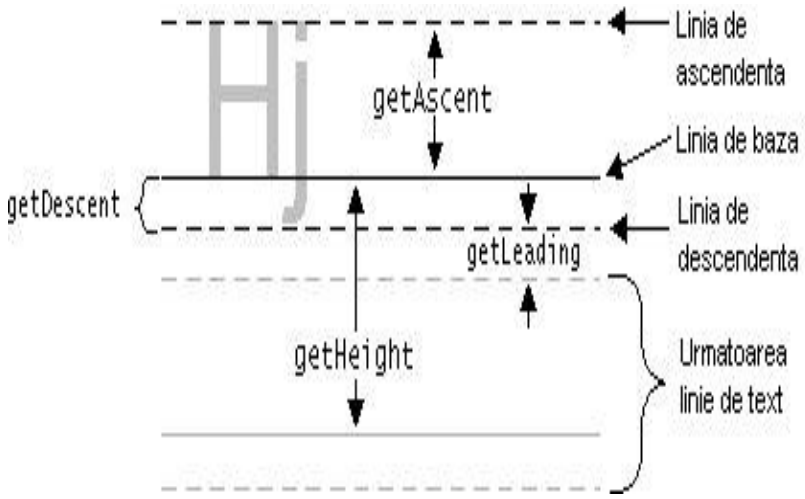
La afişarea unui şir cu metoda `drawString` trebuie să specificăm poziţia la care să apară textul respectiv pe ecran. In momentul în care avem de afişat mai multe şiruri consecutiv, sau unele sub altele, trebuie să calculăm poziţiile lor de afişare în funcţie de lungimea şi înalţimea în pixeli a celorlalte texte. Pentru aceasta este folosită clasa **FontMetrics**. Un obiect din această clasă se construieşte pornind de la un obiect de tip `Font` şi pune la dispoziţie informaţii despre dimensiunile în pixeli pe care le au caracterele fontului respectiv într-un anumit context de desenare.

Aşadar, un obiect de tip `FontMetrics` încapsulează informaţii despre *metrica* unui font, cu alte cuvinte despre dimensiunile în pixeli ale caracterelor sale. Utilitatea principală a acestei clase constă în faptul că permite poziţionarea precisă a textelor pe o suprafaţă de desenare, indiferent de fontul folosit de acestea.

Metrica unui font constă în următoarele atribute pe care le au caracterele sale:

- Linia de bază: este linia după care sunt aliniate caracterele unui font;
- Linia de ascendenţă: linia superioara pe care nu o depaseste nici un caracter din font
- Linia de descendenţă: linia inferioară sub care nu coboară nici un caracter din font;
- Ascendentul: distanţa între linia de bază şi linia de ascendenţă;
- Descendentul: distanţa între linia de bază şi linia de descendenţă;
- Lăţimea: lăţimea unui anumit caracter din font;
- Distanţa între linii ("leading"): distanţa optimă între două linii de text scrise cu acelaşi font.
- Inălţimea: distanţa dintre liniile de bază (leading+ascent+descent);

Figura de mai jos prezintă o imagine reprezentativă asupra metricii unui font:



Reamintim că la metoda `drawString(String s, int x, int y)` argumentele x şi y repreznită colţul din **stânga-jos** al textului. Ca să fim mai precişi, y reprezintă poziţia liniei de bază a textului care va fi scris.

Un context grafic pune la dispoziţie o metodă specială `getFontMetrics` de creare a unui obiect de tip `FontMetrics`, pornind de la fontul curent al contextului grafic:

```
public void paint(Graphics g) {
  Font f = new Font("Arial", Font.BOLD, 11);
  FontMetrics fm = g.getFontMetrics();
}
```

Cele mai folosite metode ale clasei `FontMetrics` sunt:

- `getHeight` - determină înălţimea unei linii pe care vor fi scrise caractere ale unui font;
- `stringWidth` - determină lăţimea totală în pixeli a unui şir de caractere specificat;
- `charWidth` - determină lăţimea unui anumit caracter din font.

In exemplul următor sunt afişate pe ecran zilele săptămânii şi lunile anului:

Listing 10.4: Folosirea clasei FontMetrics

```
import java.awt.*;

class Texte extends Canvas {
  Dimension canvasSize = new Dimension(800, 100);
  private String[] zile= {
    "Luni", "Marti", "Miercuri", "Joi",
    "Vineri", "Sambata", "Duminica"};
  private String[] luni = {
    "Ianuarie", "Februarie", "Martie", "Aprilie",
    "Mai", "Iunie", "Iulie", "August", "Septembrie",
    "Octombrie", "Noiembrie", "Decembrie"};

  public void paint(Graphics g) {
    FontMetrics fm;
    int x,y;
    String etZile = "Zilele saptamanii:",
           etLuni = "Lunile anului:", text;

    // Alegem un font si aflam metrica sa
    g.setFont(new Font("Arial", Font.BOLD, 20));
    fm = g.getFontMetrics();
    x = 0;
    y = fm.getHeight();
    g.drawString(etZile, x, y);
    x += fm.stringWidth(etZile);

    for(int i=0; i < zile.length; i++) {
      text = zile[i];
      if (i < zile.length - 1)
        text += ", ";
      g.drawString(text, x, y);
      x += fm.stringWidth(text);
    }
    // Schimbam fontul
    g.setFont(new Font("Dialog", Font.PLAIN, 14));
    fm = g.getFontMetrics();
    x = 0;
    y += fm.getHeight();
    g.drawString(etLuni, x, y);
    x += fm.stringWidth(etLuni);

    for(int i=0; i < luni.length; i++) {
      text = luni[i];
      if (i < luni.length - 1)
```

```
        text += ", ";
      g.drawString(text, x, y);
      x += fm.stringWidth(text);
    }
  }
  public Dimension getPreferredSize() {
    return canvasSize;
  }
}

class Fereastra extends Frame {
  public Fereastra(String titlu) {
    super(titlu);
    add(new Texte(),BorderLayout.CENTER);
    pack();
  }
}
public class TestFontMetrics {
  public static void main(String args[]) {
    Fereastra f = new Fereastra("FontMetrics");
    f.show();
  }
}
```

10.4 Folosirea culorilor

Orice culoare este formată prin combinaţia culorilor standard roşu (red), verde (green) şi albastru (blue), la care se adaugă un anumit grad de transparenţă (alpha). Fiecare din aceşti patru parametri poate varia într-un interval cuprins fie între 0 şi 255 (dacă dorim să specificăm valorile prin numere întregi), fie între 0.0 şi 1.0 (dacă dorim să specificăm valorile prin numere reale).

O culoare este reprezentată printr-o instanţă a clasei **Color** sau a subclasei sale **SystemColor**. Pentru a crea o culoare avem două posibilităţi:

- Să folosim una din constantele definite în cele două clase;
- Să folosim unul din constructorii clasei `Color`.

Să vedem mai întâi care sunt constantele definite în aceste clase:

Color	**SystemColor**
black	activeCaption
blue	activeCaptionBorder
cyan	activeCaptionText
darkGray	control
gray	controlHighlight
green	controlShadow
lightGray	contolText
magenta	desktop
orange	menu
pink	text
red	textHighlight
white	window
yellow	...

Observaţi că în clasa `Color` sunt definite culori uzuale din paleta standard de culori, în timp ce în clasa `SystemColor` sunt definite culorile componentelor standard (ferestre, texte, meniuri, etc) ale platformei curente de lucru. Folosirea acestor constante se face ca în exemplele de mai jos:

```
Color rosu = Color.red;
Color galben = Color.yellow;
Color fundal = SystemColor.desktop;
```

Dacă nici una din aceste culori predefinite nu corespunde preferinţelor noastre, atunci putem crea noi culori prin intermediul constructorilor clasei `Color`:

```
Color(float red, flot green,  float blue)
Color(flot red, float green,  float blue, float alpha)
Color(int red, int green,  int blue)
Color(int red, int green,  int blue, int alpha)
Color(int rgb)
```

unde *red*, *green*, *blue*, *alpha* sunt valorile pentru roşu, verde, albastru şi transparenţă iar parametrul "rgb" de la ultimul constructor reprezintă un întreg format din: biţii 16-23 roşu, 8-15 verde, 0-7 albastru.

Valorile argumentelor variază între $0 - 255$ pentru tipul `int`, respectiv $0.0 - 1.0$ pentru tipul `float`. Valoarea 255 (sau 1.0) pentru transparenţă specifică faptul că respectiva culoare este complet opacă, iar valoarea 0 (sau 0.0) specifică transparenţă totală. Implicit, culorile sunt complet opace.

```
// Exemple de folosire a constructorilor:
Color alb = new Color(255, 255, 255);
Color negru = new Color(0, 0, 0);
Color rosu = new Color(255, 0, 0);
Color rosuTransparent = new Color(255, 0, 0, 128);
```

Metodele cele mai folosite ale clasei `Color` sunt:

`brighter` `darker`	Creează o noua versiune a culorii curente mai deschisă, respectiv mai închisă

`getRed`
`getGreen`
`getBlue`
`getAlpha`
	Determină parametrii din care
	este alcatuită culoarea

`getRGB`	Determină valoarea ce reprezintă culoarea respectivă (biţii 16-23 roşu, 8-15 verde, 0-7 albastru)

Să considerăm o aplicaţie cu ajutorul căreia putem vizualiza dinamic culorile obţinute prin diferite combinaţii ale parametrilor ce formează o culoare. Aplicaţia va arăta astfel:

Listing 10.5: Folosirea clasei Color

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

class Culoare extends Canvas {
  public Color color = new Color(0, 0, 0, 255);
  Dimension canvasSize = new Dimension(150, 50);

  public void paint(Graphics g) {
    g.setColor(Color.black);
```

```
    g.setFont(new Font("Arial", Font.BOLD, 12));
    String text = "";
    text += " R=" + color.getRed();
    text += " G=" + color.getGreen();
    text += " B=" + color.getBlue();
    text += " A=" + color.getAlpha();
    g.drawString(text, 0, 30);

    g.setColor(color);
    g.fillRect(0, 0, canvasSize.width, canvasSize.height);
  }

  public Dimension getPreferredSize() {
    return canvasSize;
  }
}

class Fereastra extends Frame implements AdjustmentListener {
  private Scrollbar rValue, gValue, bValue, aValue;
  private Culoare culoare;

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

    Panel rgbValues = new Panel();
    rgbValues.setLayout(new GridLayout(4, 1));
    rValue = new Scrollbar(Scrollbar.HORIZONTAL, 0, 1, 0,
       256);
    rValue.setBackground(Color.red);

    gValue = new Scrollbar(Scrollbar.HORIZONTAL, 0, 1, 0,
       256);
    gValue.setBackground(Color.green);

    bValue = new Scrollbar(Scrollbar.HORIZONTAL, 0, 1, 0,
       256);
    bValue.setBackground(Color.blue);

    aValue = new Scrollbar(Scrollbar.HORIZONTAL, 0, 1, 0,
       256);
```

```
    aValue.setValue(255);
    aValue.setBackground(Color.lightGray);

    rgbValues.add(rValue);
    rgbValues.add(gValue);
    rgbValues.add(bValue);
    rgbValues.add(aValue);
    rgbValues.setSize(200, 100);
    add(rgbValues, BorderLayout.CENTER);

    culoare = new Culoare();
    add(culoare, BorderLayout.NORTH);

    pack();

    rValue.addAdjustmentListener(this);
    gValue.addAdjustmentListener(this);
    bValue.addAdjustmentListener(this);
    aValue.addAdjustmentListener(this);
  }

  public void adjustmentValueChanged(AdjustmentEvent e) {
    int r = rValue.getValue();
    int g = gValue.getValue();
    int b = bValue.getValue();
    int a = aValue.getValue();
    Color c = new Color(r, g, b, a);
    culoare.color = c;
    culoare.repaint();
  }
}

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

10.5 Folosirea imaginilor

Aceasta este o imagine:

In AWT este posibilă folosirea imaginilor create extern în format *gif* sau *jpeg*. Orice imagine va fi reprezentată ca o instanţă a clasei **Image**. Aceasta nu este o clasă de componente (nu extinde `Component`) ci implementează obiecte care pot fi desenate pe suprafaţa unor componente cu metode specifice unui context grafic pentru componenta respectiva (similar modului cum se desenează o linie sau un cerc).

10.5.1 Afişarea imaginilor

Afişarea unei imagini presupune realizarea următoarilor doi paşi:

1. Crearea unui obiect de tip `Image`;

2. Afişarea propriu-zisă într-un context grafic;

Crearea unui obiect de tip `Image` se face folosind o imagine dintr-un fişier fie aflat pe maşina pe care se lucrează, fie aflat la o anumită adresă Web (URL). Metodele pentru încărcarea unei imagini dintr-un fişier se găsesc în clasele `Applet` şi `Toolkit`, având însă aceeaşi denumire **getImage** şi următoarele formate:

Applet	**Toolkit**
`getImage(URL url)`	`getImage(URL url)`
`getImage(URL url, String fisier)`	`getImage(String fisier)`

Pentru a obţine un obiect de tip `Toolkit` se va folosi metoda `getDefaultToolkit`, ca în exemplul de mai jos:

```
Toolkit toolkit = Toolkit.getDefaultToolkit();
Image image1 = toolkit.getImage("poza.gif");
Image image2 = toolkit.getImage(
  new URL("http://www.infoiasi.ro/~acf/poza.gif"));
```

Metoda `getImage` nu verifică dacă fişierul sau adresa specificata reprezintă o imagine validă şi nici nu încarcă efectiv imaginea în memorie, aceste operaţiuni fiind făcute abia în momentul în care se va realiza afişarea imaginii pentru prima dată. Metoda nu face decât să creeze un obiect de tip `Image` care face referinţă la o anumită imagine externă.

Afişarea unei imagini într-un context grafic se realizează prin intermediul metodei **drawImage** din clasa `Graphics` şi, în general, va fi facută în metoda `paint` a unei componente. Cele mai uzuale formate ale metodei sunt:

```
boolean drawImage(Image img, int x, int y, ImageObserver observer)
boolean drawImage(Image img, int x, int y, Color bgcolor,
    ImageObserver observer)
boolean drawImage(Image img, int x, int y, int width, int height,
    ImageObserver observer)
boolean drawImage(Image img, int x, int y, int width, int height,
    Color bgcolor, ImageObserver observer)
```

unde:

- *img* este obiectul ce reprezintă imaginea;
- x, y sunt coordonatele stânga-sus la care va fi afişată imaginea, relative la spaţiul de coordonate al contextului grafic;
- *observer* este un obiect care "observă" încărcarea imaginii şi va fi informat pe măsura derulării acesteia;
- $width, heigth$ reprezintă înalţimea şi lăţimea la care trebuie scalată imaginea (dacă lipsesc, imaginea va fi afişată la dimensiunile ei reale);
- *bgColor* reprezintă culoarea cu care vor fi coloraţi pixelii transparenţi ai imaginii (poate să lipsească).

In exemplul următor afişam aceeaşi imagine de trei ori, folosind forme diferite ale metodei `drawImage`:

```
Image img = Toolkit.getDefaultToolkit().getImage("taz.gif");
g.drawImage(img, 0, 0, this);
g.drawImage(img, 0, 200, 100, 100, this);
g.drawImage(img, 200, 0, 200, 400, Color.yellow, this);
```

Metoda `drawImage` returnează `true` dacă imaginea a fost afişată în întregime şi `false` în caz contrar, cu alte cuvinte metoda nu aşteptă ca o imagine să fie complet afişată ci se termină imediat ce procesul de afişare a început. In secţiunea următoare vom detalia acest aspect.

10.5.2 Monitorizarea încărcării imaginilor

In cazul în care se afişează o imagine care se găseşte pe Internet sau imaginea afişata este de dimensiuni mari se va observa că aceasta nu apare complet de la început ci este desenată treptat, fără intervenţia programatorului. Acest lucru se întâmplă deoarece metoda `drawImage` nu face decât să declanşeze procesul de încărcare şi desenare a imaginii, după care redă imediat controlul apelantului, lucru deosebit de util întrucât procesul de încărcare a unei imagini poate dura mult şi nu este de dorit ca în acest interval de timp (până la încărcarea completă a imaginii) aplicaţia să fie blocată. Ca urmare, la apelul metodei `drawImage` va fi desenată numai porţiunea de imagine care este disponibilă la momentul iniţial şi care poate fi incompletă. De aceea trebuie să existe un mecanism prin care componenta să fie redesenată automat în momentul în care au mai sosit informaţii legate de imagine, până la afişarea sa completă. Acest mecanism este realizat prin intermediul interfeţei **ImageObserver**, implementată de clasa `Component` şi deci de toate componentele. Această interfaţă descrie obiecte care au început să utilizeze o imagine incompletă şi care trebuie anunţate de noile date obţinute în legatură cu imaginea respectivă.

Interfaţa `ImageObserver` are o singură metodă numită **imageUpdate**, ce va fi apelată periodic de firul de execuţie (creat automat) care se ocupă cu încărcarea imaginii. Formatul acestei metode este:

```
boolean imageUpdate (Image img, int flags,
                     int x, int y, int w, int h )
```

Implementarea implicită constă dintr-un apel la metoda `repaint` pentru dreptunghiul specificat la apel şi care reprezintă zona din imagine pentru care se cunosc noi informaţii. Intregul *flags* furnizează informaţii despre starea transferului. Aceste informaţii pot fi aflate prin intermediul constantelor definite de interfaţă:

ABORT	Incărcarea imaginii a fost întreruptă, înainte de completarea sa
ALLBITS	Imaginea a fost încarcată complet
ERROR	A apărut o eroare în timpul încărcării imaginii
FRAMEBITS	Toţi biţii cadrului curent sunt disponibili
HEIGHT	Inălţimea imaginii este disponibilă
PROPERTIES	Proprietăţile imaginii sunt disponibile
SOMEBITS	Au fost recepţionaţi noi pixeli ai imaginii
WIDTH	Lăţimea imaginii este disponibilă

Prezenţa în parametrul $flags$ a unui bit de valoare 1 pe poziţia reprezentata de o constantă înseamnă că respectiva condiţie este îndeplinită.

```
// Imaginea este completa
(flags & ALLBITS) != 0

// Eroare sau transferul imaginii a fost intrerupt
(flags & ERROR | ABORT ) != 0
```

Metoda `imageUpdate` poate fi redefintă de o componentă pentru a personaliza procesul de afişare al imaginii. Aceasta va fi apelată apelată asincron de fiecare dată când sunt disponibili noi pixeli.

```
  public boolean imageUpdate(Image img, int flags,
      int x, int y, int w, int h) {
    // Desenam imaginea numai daca toti bitii sunt disponibili
    if (( flags & ALLBITS) != 0)
      repaint();

    // Daca sunt toti bitii nu mai sunt necesare noi update-uri
    return ( (flags & (ALLBITS | ABORT)) == 0);
}
```

De asemenea, se observă că metodele clasei `Image` pentru determinarea dimensiunilor unei imagini au ca argument un obiect de tip `ImageObserver`.

```
  int getHeight(ImageObserver observer)
  int getWidth(ImageObserver observer)
```

Dacă desenarea se face folosind clasa `Canvas`, atunci argumentul *observer* al metodelor referitoare la imagini va fi **this**.

10.5.3 Mecanismul de "double-buffering"

Tehnica de double-buffering implică realizarea unui desen în memorie şi apoi transferul său pe ecran, pentru a elimina efectul neplăcut de "clipire" ("flickering") rezultat atunci când sunt efectuate redesenări repetate la intervale mici de timp. O situaţie frecventă în care se apelează la double-buffering este crearea de animaţii.

Secvenţa generală de implementare a mecanismului de double-buffering este următoarea:

```
  // Supradefinim update pentru a elimina stergerea desenului
  public void update(Graphics g) {
    paint(g);
  }

  public void paint(Graphics g) {
    // Desenam in memorie pe un obiect de tip Image
    // w si h sunt dimensiunile desenului
    Image img = createImage(w, h);
    Graphics gmem = img.getGraphics();

    /* Realizam desenul folosind gmem
      gmem.setColor(...);
      gmem.fillOval(...);
      ...
    */

    // Transferam desenul din memorie pe ecran
    // desenand de fapt imaginea creata
    g.drawImage(img, 0, 0, this);

    gmem.dispose();
  }
}
```

10.5.4 Salvarea desenelor în format JPEG

Pachetul `com.sun.image.codec.jpeg` din distribuţia standard Java oferă suport pentru salvarea unei imagini aflate în memorie într-un fişier în for-

mat JPEG. O clasă responsabilă cu realizarea acestei operaţiuni ar putea fi definită astfel:

```
import com.sun.image.codec.jpeg.*;
import java.awt.image.BufferedImage;
import java.awt.*;
import java.io.*;

class JPEGWriter {

  static float quality = 0.9f; //intre 0 si 1
  public static void write(BufferedImage img, String filename) {

    try {
      FileOutputStream out = new FileOutputStream(filename);

      JPEGImageEncoder encoder =
          JPEGCodec.createJPEGEncoder(out);
      JPEGEncodeParam jep =
          encoder.getDefaultJPEGEncodeParam(img);
      jep.setQuality(quality, false);

      // Folosim setarile de codare jpeg implicite
      encoder.setJPEGEncodeParam(jep);
      encoder.encode(img);

      out.close();
    } catch( Exception e ) {
      e.printStackTrace();
    }
  }
}
```

10.5.5 Crearea imaginilor în memorie

In cazul în care dorim să folosim o anumită imagine creată direct din program şi nu încarcată dintr-un fişier vom folosi clasa **MemoryImageSource**, aflata în pachetul `java.awt.image`. Pentru aceasta va trebui să definim un

vector de numere întregi în care vom scrie valorile întregi (RGB) ale culorilor pixelilor ce definesc imaginea noastră. Dimensiunea vectorului va fi înălţimea înmulţită cu lăţimea în pixeli a imaginii. Constructorul clasei `MemoryImageSource` este:

```
MemoryImageSource(int w, int h, int[] pixeli, int off, int scan)
```

unde:

- w, h reprezintă dimensiunile imaginii (lăţimea şi înălţimea);
- $pixeli[]$ este vectorul cu culorile imaginii;
- $off, scan$ reprezintă modalitatea de construire a matricii imaginii pornind de la vectorul cu pixeli, normal aceste valori sunt off = 0, scan = w

In exemplul urmator vom crea o imagine cu pixeli de culori aleatorii şi o vom afişa pe ecran:

```
int w = 100;
int h = 100;
int[] pix = new int[w * h];
int index = 0;
for (int y = 0; y < h; y++) {
for (int x = 0; x < w; x++) {
  int red = (int) (Math.random() * 255);
  int green = (int) (Math.random() * 255);
  int blue = (int) (Math.random() * 255);
  pix[index++] = new Color(red, green, blue).getRGB();
}

img = createImage(new MemoryImageSource(w, h, pix, 0, w));
g.drawImage(img, 0, 0, this);  // g este un context grafic
```

10.6 Tipărirea

Tipărirea în Java este tratată în aceeaşi manieră ca şi desenarea, singurul lucru diferit fiind contextul grafic în care se execută operaţiile. Pachetul care oferă suport pentru tipărire este **java.awt.print**, iar clasa principală care controlează procesul de tipărire este **PrinterJob**. O aplicaţie va apela metode ale acestei clase pentru:

- Crearea unei sesiuni de tipărire (job);
- Invocarea dialogului cu utilizatorul pentru specificarea unor parametri legaţi de tipărire;
- Tipărirea efectivă.

Orice componentă care poate fi afişată pe ecran poate fi şi tipărită. In general, orice informaţii care trebuie atât afişate cât şi tipărite, vor fi încapsulate într-un obiect grafic - componentă, care are o reprezentare vizuală descrisă de metoda `paint` şi care va specifica şi modalitatea de reprezentare a sa la imprimantă.

Un obiect care va fi tipărit trebuie să implementeze interfaţa **Printable**, care conţine o singură metodă **print**, responsabilă cu descrierea modalităţii de tipărire a obiectului. In cazul când imaginea de pe ecran coincide cu imaginea de la imprimantă, metodele `paint` şi `print` pot specifica aceeaşi secvenţă de cod. In general, metoda `print` are următorul format:

```
  public int print(Graphics g, PageFormat pf, int pageIndex)
throws PrinterException {

    // Descrirea imaginii obiectului ce va fi afisata la imprimanta
    // Poate fi un apel la metoda paint: paint(g)

    if (ceva nu este in regula}) {
      return Printable.NO_SUCH_PAGE;
    }

    return Printable.PAGE_EXISTS;
  }
```

Paşii care trebuie efectuaţi pentru tipărirea unui obiect sunt:

1. Crearea unei sesiuni de tipărire: `PrinterJob.getPrinterJob`
2. Specificarea obiectului care va fi tipărit: `setPrintable`; acesta trebuie să implementeze interfaţa `Printable`;
3. Opţional, iniţierea unui dialog cu utilizatorul pentru precizarea unor parametri legaţi de tipărire: `printDialog`;

4. Tipărirea efectivă: `print`.

In exemplul următor vom defini un obiect care are aceeaşi reprezentare pe ecran cât şi la imprimantă (un cerc circumscris unui pătrat, însotite de un text) şi vom tipări obiectul respectiv.

Listing 10.6: Tipărirea unei componente

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

class Plansa extends Canvas implements Printable {
  Dimension d = new Dimension(400, 400);
  public Dimension getPreferredSize() {
    return d;
  }

  public void paint(Graphics g) {
    g.drawRect(200, 200, 100, 100);
    g.drawOval(200, 200, 100, 100);
    g.drawString("Hello", 200, 200);
  }

  public int print(Graphics g, PageFormat pf, int pi)
      throws PrinterException {
    if (pi >= 1)
      return Printable.NO_SUCH_PAGE;

    paint(g);
    g.drawString("Numai la imprimanta", 200, 300);

    return Printable.PAGE_EXISTS;
  }
}

class Fereastra extends Frame implements ActionListener {
  private Plansa plansa = new Plansa();
  private Button print = new Button("Print");

  public Fereastra(String titlu) {
    super(titlu);
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
```

```
        System.exit(0);
      }
    });

    add(plansa, BorderLayout.CENTER);

    Panel south = new Panel();
    south.setLayout(new FlowLayout(FlowLayout.CENTER));
    south.add(print);
    add(south, BorderLayout.SOUTH);

    print.addActionListener(this);
    pack();
  }

  public void actionPerformed(ActionEvent e) {
    // 1. Crearea unei sesiuni de tiparire
    PrinterJob printJob = PrinterJob.getPrinterJob();

    // 2. Stabilirea obiectului ce va fi tiparit
    printJob.setPrintable(plansa);

    // 3. Initierea dialogului cu utilizatorul
    if (printJob.printDialog()) {
      try {
        // 4. Tiparirea efectiva
        printJob.print();
      } catch (PrinterException ex) {
        System.out.println("Exceptie la tiparire!");
        ex.printStackTrace();
      }
    }
  }
}

public class TestPrint {
  public static void main(String args[]) throws Exception {
    Fereastra f = new Fereastra("Test Print");
    f.show();
  }
}
```

Tiparirea textelor

O altă variantă pentru tipărirea de texte este deschiderea unui flux către dispozitivul special reprezentat de imprimantă şi scrierea informaţiilor, linie cu linie, pe acest flux. In sistemul de operare Windows, imprimanta poate fi referită prin **"lpt1"**, iar în Unix prin **"/dev/lp"**. Observaţi că această abordare nu este portabilă, deoarece necesită tratare specială în funcţie de sistemul de operare folosit.

Listing 10.7: Tipărirea textelor

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

class TestPrintText {
  public static void main(String args[]) throws Exception {
    // pentru Windows
    PrintWriter imp = new PrintWriter(new FileWriter("lpt1"))
        ;

    // pentru UNIX
    //PrintWriter imp = new PrintWriter(new FileWriter("/dev/
        lp"));

    imp.println("Test imprimanta");
    imp.println("ABCDE");
    imp.close();
  }
}
```

Capitolul 11

Swing

11.1 Introducere

11.1.1 JFC

Tehnologia Swing face parte dintr-un proiect mai amplu numit **JFC** (Java Foundation Classes) care pune la dispoziţie o serie întreagă de facilităţi pentru scrierea de aplicaţii cu o interfaţă grafică mult îmbogăţită funcţional şi estetic faţă de vechiul model AWT. In JFC sunt incluse următoarele:

- **Componente Swing**
  Sunt componente ce înlocuiesc şi în acelaşi timp extind vechiul set oferit de modelul AWT.

- **Look-and-Feel**
  Permite schimbarea înfăţişării şi a modului de interacţiune cu aplicaţia în funcţie de preferinţele fiecăruia. Acelaşi program poate utiliza diverse moduri Look-and-Feel, cum ar fi cele standard Windows, Mac, Java, Motif sau altele oferite de diverşi dezvoltatori, acestea putând fi interschimbate de către utilizator chiar la momentul execuţiei .

- **Accessibility API**
  Permite dezvoltarea de aplicaţii care să comunice cu dispozitive utilizate de către persoane cu diverse tipuri de handicap, cum ar fi cititoare de ecran, dispozitive de recunoaştere a vocii, ecrane Braille, etc.

- **Java 2D API**
  Folosind Java 2D pot fi create aplicaţii care utilizează grafică la un

nivel avansat. Clasele puse la dispoziţie permit crearea de desene complexe, efectuarea de operaţii geometrice (rotiri, scalări, translaţii, etc.), prelucrarea de imagini, tipărire, etc.

- **Drag-and-Drop**
  Oferă posibilitatea de a efectua operaţii drag-and-drop între aplicaţii Java şi aplicaţii native.

- **Internaţionalizare**
  Internaţionalizarea şi localizarea aplicaţiilor sunt două facilităţi extrem de importante care permit dezvoltarea de aplicaţii care să poată fi configurate pentru exploatarea lor în diverse zone ale globului, utilizând limba şi particularităţile legate de formatarea datei, numerelor sau a monedei din zona respectivă.

In aceste capitol vom face o prezentare scurtă a componentelor Swing, deoarece prezentarea detaliata a tuturor facilităţilor oferite de JFC ar oferi suficient material pentru un volum de sine stătător.

11.1.2 Swing API

Unul din principalele deziderate ale tehnologiei Swing a fost să pună la dispoziţie un set de componente GUI extensibile care să permită dezvoltarea rapidă de aplicaţii Java cu interfaţă grafică competitivă din punct de vedere comercial. Pentru a realiza acest lucru, API-ul oferit de Swing este deosebit de complex având 17 pachete în care se găsesc sute de clase şi interfeţe. Lista completă a pacehetelor din distribuţia standard 1.4 este dată în tabelul de mai jos:

```
javax.accessibility      javax.swing.plaf
javax.swing.text.html    javax.swing
javax.swing.plaf.basic   javax.swing.text.parser
javax.swing.border       javax.swing.plaf.metal
javax.swing.text.rtf     javax.swing.colorchooser
javax.swing.plaf.multi   javax.swing.tree
javax.swing.event        javax.swing.table
javax.swing.undo         javax.swing.filechooser
javax.swing.text
```

Evident, nu toate aceste pachete sunt necesare la dezvolatarea unei aplicaţii, cel mai important şi care conţine componentele de bază fiind **javax.swing**.

Componentele folosite pentru crearea interfeţelor grafice Swing pot fi grupate astfel:

- **Componente atomice**
  `JLabel, JButton, JCheckBox, JRadioButton, JToggleButton, JScrollBar, JSlider, JProgressBar, JSeparator`
- **Componente complexe**
  `JTable, JTree, JComboBox, JSpinner, JList, JFileChooser, JColorChooser, JOptionPane`
- **Componente pentru editare de text**
  `JTextField, JFormattedTextField, JPasswordField, JTextArea, JEditorPane, JTextPane`
- **Meniuri**
  `JMenuBar, JMenu, JPopupMenu, JMenuItem, JCheckboxMenuItem, JRadioButtonMenuItem`
- **Containere intermediare**
  `JPanel, JScrollPane, JSplitPane, JTabbedPane, JDesktopPane, JToolBar`
- **Containere de nivel înalt**
  `JFrame, JDialog, JWindow, JInternalFrame, JApplet`

11.1.3 Asemănări şi deosebiri cu AWT

Nu se poate spune că Swing înlocuieşte modelul AWT ci îl extinde pe acesta din urmă adăugându-i noi componente care fie înlocuiesc unele vechi fie sunt cu totul noi. O convenţie în general respectată este prefixarea numelui unei clase AWT cu litera "**J**" pentru a denumi clasa corespondentă din Swing. Astfel, în locul clasei `java.awt.Button` putem folosi `javax.swing.JButton`, în loc de `java.awt.Label` putem folosi `javax.swing.JLabel`, etc. Este recomandat ca o aplicaţie cu interfaţă grafică să folosească fie componente AWT, fie Swing, amestecarea lor fiind mai puţin uzuală.

Aplicaţiile GUI vor avea în continuare nevoie de pachetul `java.awt` deoarece aici sunt definite unele clase utilitare cum ar fi `Color, Font, Dimension`, etc. care nu au fost rescrise în Swing.

De asemenea, pachetul `java.awt.event` rămâne în continuare esenţial pentru tratarea evenimentelor generate atât de componente AWT cât şi de cele din Swing. Pe lângă acesta mai poate fi necesar şi `javax.swing.event` care descrie tipuri de evenimente specifice unor componente Swing, mecanismul de tratare a lor fiind însă acelaşi ca în AWT.

Poziţionarea componentelor este preluată din AWT, fiind adăugate însă noi clase care descriu gestionari de poziţionare în completarea celor existente, cum ar fi `BoxLayout` şi `SpringLayout`. Diferă însă modul de lucru cu containere, după cum vom vedea în secţiunea dedicată acestora.

Majoritatea componentelor Swing care permit afişarea unui text ca parte a reprezentării lor GUI pot specifica acel text fie în mod normal folosind un anumit font şi o anumită culoare ce pot fi setate cu metodele `setFont` şi `setColor`, fie prin intermediul limbajului HTML. Folosirea HTML aduce o flexibilitatea deosebită în realizarea interfeţei grafice, întrucât putem aplica formatări multiple unui text, descompunerea acestuia pe mai multe linii, etc., singurul dezavantaj fiind încetinirea etapei de afişare a componentelor.

```
JButton simplu = new JButton("Text simplu");
JButton html = new JButton(
  "<html><u>Text</u> <i>formatat</i></html>");
```

Să descriem o aplicţie simplă folosind AWT şi apoi Swing, pentru a ne crea o primă impresie asupra diferenţelor şi asemănărilor dintre cele două modele.

Listing 11.1: O aplicaţie simplă AWT

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

public class ExempluAWT extends Frame implements
   ActionListener {
  public ExempluAWT(String titlu) {
    super(titlu);
    setLayout(new FlowLayout());
    add(new Label("Hello AWT"));
    Button b = new Button("Close");
    b.addActionListener(this);
```

```
    add(b);
    pack();
    show();
  }
  public void actionPerformed(ActionEvent e) {
    System.exit(0);
  }
  public static void main(String args[]) {
    new ExempluAWT("Hello");
  }
}
```

Listing 11.2: Aplicaţia rescrisă folosind Swing

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

public class ExempluSwing extends JFrame implements
   ActionListener {
  public ExempluSwing(String titlu) {
    super(titlu);
    // Metoda setLayout nu se aplica direct ferestrei
    getContentPane().setLayout(new FlowLayout());

    // Componentele au denumiri ce incep cu litera J
    // Textul poate fi si in format HTML
    getContentPane().add(new JLabel(
      "<html><u>Hello</u> <i>Swing</i></html>"));
    JButton b = new JButton("Close");
    b.addActionListener(this);

    // Metoda add nu se aplica direct ferestrei
    getContentPane().add(b);
    pack();
    show();
  }
  public void actionPerformed(ActionEvent e) {
    // Tratarea evenimentelor se face ca in AWT
    System.exit(0);
  }
  public static void main(String args[]) {
    new ExempluSwing("Hello");
  }
```

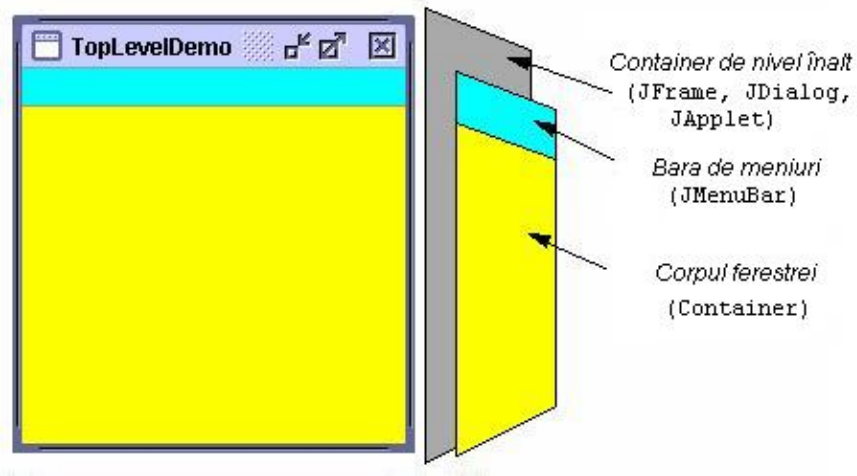
```
}
```

11.2 Folosirea ferestrelor

Pentru a fi afişate pe ecran componentele grafice ale unei aplicaţii trebuie plasate pe o suprafaţă de afişare (container). Fiecare componentă poate fi conţinută doar într-un singur container, adăugarea ei pe o suprafţă nouă de afişare determinând eliminarea ei de pe vechiul container pe care fusese plasată. Intrucât containerele pot fi încapsulate în alte containere, o componentă va face parte la un moment dat dintr-o ierarhie. Rădăcina acestei ierarhii trebuie să fie un aşa numit *container de nivel înalt*, care este reprezentat de una din clasele **JFrame**, **JDialog** sau **JApplet**. Intrucât de appleturi ne vom ocupa separat, vom analiza în continuare primele două clase.

In general orice aplicaţie Java independentă bazată pe Swing conţine cel puţin un container de nivel înalt reprezentat de fereastra principală a programului, instanţă a clasei `JFrame`.

Simplificat, un obiect care reprezintă o fereastră Swing conţine o zonă care este rezervată barei de meniuri şi care este situată de obieci în partea sa superioară şi corpul ferestrei pe care vor fi plasate componentele. Imaginea de mai jos pune în evidenţă această separare, valabilă de altfel pentru orice container de nivel înalt:



Corpul ferestrei este o instanţă a clasei `Container` ce poate fi obţinută cu metoda **getContentPane**. Plasarea şi aranjarea componentelor pe suprafaţa

ferestrei se va face deci folosind obiectul de tip `Container` şi nu direct fereastra. Aşadar, deşi este derivată din `Frame`, clasa `JFrame` este folosită într-un mod diferit faţă de părintele său:

```
Frame f = new Frame();
f.setLayout(new FlowLayout());
f.add(new Button("OK"));

JFrame jf = new JFrame();
jf.getContentPane().setLayout(new FlowLayout());
jf.getContentPane().add(new JButton("OK"));
```

Spre deosebire de `Frame`, un obiect `JFrame` are un comportament implicit la închiderea ferestrei care constă în ascunderea ferestrei atunci când utilizatorul apasă butonul de închidere. Acest comportament poate fi modificat prin apelarea metodei **setDefaultCloseOperation** care primeşte ca argument diverse constante ce se găsesc fie în clasa `WindowConstants`, fie chiar în `JFrame`.

```
jf.setDefaultCloseOperation(WindowConstants.HIDE_ON_CLOSE);
jf.setDefaultCloseOperation(WindowConstants.DO_NOTHING_ON_CLOSE);
jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Adăugarea unei bare de meniuri se realizează cu metoda **setJMenuBar**, care primeşte o instanţă de tip `JMenuBar`. Crearea meniurilor este similară cu modelul AWT.

11.2.1 Ferestre interne

Din punctul de vedere al folosirii ferestrelor, aplicaţiile pot fi împărţite în două categorii:

- *SDI (Single Document Interface)*
- *MDI (Multiple Document Interface)*

Programele din prima categorie gestionează la un moment dat o singură fereastră în care se găsesc componentele cu care interacţionează utilizatorul. In a doua categorie, fereastra principală a aplicaţiei înglobează la rândul ei alte ferestre, uzual cu funcţionalităţi similare, ce permit lucrul concurent pe mai multe planuri.

In Swing, clasa **JInternalFrame** pune la dispoziţie o modalitate de a crea ferestre în cadrul altor ferestre. Ferestrele interne au aproximativ aceeaşi înfăţişare şi funcţionalitate cu ferestrele de tip `JFrame`, singura diferenţă fiind modul de gestionare a acestora.

Uzual, obiectele de tip `JInternalFrame` vor fi plasate pe un container de tip **DesktopPane**, care va fi apoi plasat pe o fereastră de tip `JFrame`. Folosirea clasei `DesktopPane` este necesară deoarece aceasta "ştie" cum să gestioneze ferestrele interne, având în vedere că acestea se pot suprapune şi la un moment dat doar una singură este activă.

Exemplul următor pune în evidenţă modelul general de creare şi afişare a ferestrelor interne:

Listing 11.3: Folosirea ferestrelor interne

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

class FereastraPrincipala extends JFrame {
    public FereastraPrincipala(String titlu) {
      super(titlu);
      setSize(300, 200);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      FereastraInterna fin1 = new FereastraInterna();
      fin1.setVisible(true);
      FereastraInterna fin2 = new FereastraInterna();
      fin2.setVisible(true);

      JDesktopPane desktop = new JDesktopPane();
      desktop.add(fin1);
      desktop.add(fin2);
      setContentPane(desktop);

      fin2.moveToFront();
  }
}

class FereastraInterna extends JInternalFrame {
  static int n = 0; //nr. de ferestre interne
  static final int x = 30, y = 30;

  public FereastraInterna() {
    super("Document #" + (++n),
```

```
            true, //resizable
            true, //closable
            true, //maximizable
            true);//iconifiable
      setLocation(x*n, y*n);
      setSize(new Dimension(200, 100));
    }
}
public class TestInternalFrame {
  public static void main(String args[]) {
    new FereastraPrincipala("Test ferestre interne").show();
  }
}
```

Ferestrele create de acest program vor arăta ca în figura de mai jos:

11.3 Clasa JComponent

`JComponent` este superclasa tuturor componentelor Swing, mai puţin a celor care descriu containere de nivel înalt `JFrame`, `JDialog`, `JApplet`. Deoarece `JComponent` extinde clasa `Container`, deci şi `Component`, ea moşteneşte funcţionalitatea generală a containerelor şi componentelor AWT, furnizând bineînţeles şi o serie întreagă de noi facilităţi.

Dintre noutăţile oferite de `JComponent` amintim:

- **ToolTips**
  Folosind metoda `setToolTip` poate fi ataşat unei componente un text cu explicaţii legate de componenta respectivă. Când utilizatorul trece

cu mouse-ul deasupra componentei va fi afişat, pentru o perioadă de timp, textul ajutător specificat.

- **Chenare**
  Orice componentă Swing poate avea unul sau mai multe chenare. Specificarea unui chenar se realizează cu metoda `setBorder`.

- **Suport pentru plasare şi dimensionare**
  Folosind metodele `setPreferredSize, setMinimumSize, setMaximumSize, setAlignmentX, setAlignmentY` pot fi controlaţi parametrii folosiţi de gestionarii de poziţionare pentru plasarea şi dimensionarea automată a componentelor în cadrul unui container.

- **Controlul opacităţii**
  Folosind metoda `setOpaque` vom specifica dacă o componentă trebuie sau nu să deseneze toţi pixelii din interiorul său. Implicit, valoarea proprietăţii de opacitate este `false`, ceea ce înseamnă că este posibil să nu fie desenaţi unii sau chiar toţi pixelii, permiţând pixelilor de sub componentă să rămână vizibili (componenta nu este opacă). Valoarea proprietăţii pentru clasele derivate din JComponent depinde în general de Look-and-Feel-ul folosit.

- **Asocierea de acţiuni tastelor**
  Pentru componentele Swing există posibilitatea de specifica anumite acţiuni care să se execute atunci când utilizatorul apasă o anumită combinaţie de taste şi componenta respectivă este activă (are focus-ul). Această facilitate simplifică varianta iniţială de lucru, şi anume tratarea evenimentelor de tip `KeyEvent` printr-un obiect `KeyListener`.

- **Double-Buffering**
  Tehnica de double-buffering, care implică desenarea componentei în memorie şi apoi transferul întregului desen pe ecran, este implementată automat de componentele Swing, spre deosebire de cele AWT unde trebuia realizată manual dacă era cazul.

Exemplul următor ilustrează modul de folosire a câtorva dintre facilităţile amintite mai sus:

Listing 11.4: Facilităţi oferite de clasa `JComponent`

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

class Fereastra extends JFrame {
  public Fereastra(String titlu) {
    super(titlu);
    getContentPane().setLayout(new FlowLayout());
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Folosirea chenarelor
    Border lowered, raised;
    TitledBorder title;

    lowered = BorderFactory.createLoweredBevelBorder();
    raised = BorderFactory.createRaisedBevelBorder();
    title = BorderFactory.createTitledBorder("Borders");

    final JPanel panel = new JPanel();
    panel.setPreferredSize(new Dimension(400,200));
    panel.setBackground(Color.blue);
    panel.setBorder(title);
    getContentPane().add(panel);

    JLabel label1 = new JLabel("Lowered");
    label1.setBorder(lowered);
    panel.add(label1);

    JLabel label2 = new JLabel("Raised");
    label2.setBorder(raised);
    panel.add(label2);

    // Controlul opacitatii
    JButton btn1 = new JButton("Opaque");
    btn1.setOpaque(true); // implicit
    panel.add(btn1);

    JButton btn2 = new JButton("Transparent");
    btn2.setOpaque(false);
    panel.add(btn2);

    // ToolTips
    label1.setToolTipText("Eticheta coborata");
    label2.setToolTipText("Eticheta ridicata");
```

```
    btn1.setToolTipText("Buton opac");
    // Textul poate fi HTML
    btn2.setToolTipText("<html><b>Apasati <font color=red>F2
       </font> " +
        "cand butonul are <u>focusul</u>");

    // Asocierea unor actiuni (KeyBindings)
    /* Apasarea tastei F2 cand focusul este pe butonul al
       doilea
       va determina schimbarea culorii panelului */
    btn2.getInputMap().put(KeyStroke.getKeyStroke("F2"),
                           "schimbaCuloare");
    btn2.getActionMap().put("schimbaCuloare", new
       AbstractAction() {
      private Color color = Color.red;
      public void actionPerformed(ActionEvent e) {
        panel.setBackground(color);
        color = (color == Color.red ? Color.blue : Color.red)
           ;
      }
    });

    pack();
  }
}

public class TestJComponent {
  public static void main(String args[]) {
    new Fereastra("Facilitati JComponent").show();
  }
}
```

11.4 Arhitectura modelului Swing

11.5 Folosirea modelelor

Modelul Swing este bazat pe o arhitectură asemănătoare cu *MVC (model-view-controller)*. Arhitectura MVC specifică descompunerea unei aplicaţii vizuale în trei părţi separate:

- *Modelul* - care va reprezenta datele aplicaţiei.

- *Prezentarea* - modul de reprezentare vizuală a datelor.
- *Controlul* - transformarea acţiunilor utilizatorului asupra componentelor vizuale în evenimente care să actualizeze automat modelul acestora (datele).

Din motive practice, în Swing părţile de prezentare şi control au fost cuplate deoarece exista o legătură prea strânsă între ele pentru a fi concepute ca entităţi separate. Aşadar, arhitectura Swing este de fapt o arhitectură cu *model separabil*, în care datele componentelor (modelul) sunt separate de reprezentarea lor vizuală. Această abordare este logică şi din perspectiva faptului că, în general, modul de concepere a unei aplicaţii trebuie să fie orientat asupra reprezentării şi manipulării informaţiilor şi nu asupra interfeţei grafice cu utilizatorul.

Pentru a realiza separarea modelului de prezentare, fiecărui obiect corespunzător unei clase ce descrie o componentă Swing îi este asociat un obiect care gestionează datele sale şi care implementează o interfaţă care reprezintă modelul componentei respective. După cum se observă din tabelul de mai jos, componente cu reprezentări diferite pot avea acelaşi tip de model, dar există şi componente care au asociate mai multe modele:

Model	**Componentă**
`ButtonModel`	`JButton, JToggleButton, JCheckBox, JRadioButton, JMenu, JMenuItem, JCheckBoxMenuItem, JRadioButtomMenuItem`
`JComboBox`	`ComboBoxModel`
`BoundedRangeModel`	`JProgressBar, JScrollBarm, JSlider`
`JTabbedPane`	`SingleSelectionModel`
`ListModel`	`JList`
`ListSelectionModel`	`JList`
`JTable`	`TableModel`
`JTable`	`TableColumnModel`
`JTree`	`TreeModel`
`JTree`	`TreeSelectionModel`
`Document`	`JEditorPane, JTextPane, JTextArea, JTextField, JPasswordField`

Fiecare componentă are un model iniţial implicit, însă are posibilitatea de a-l înlocui cu unul nou atunci când este cazul. Metodele care accesează

modelul unui obiect sunt: **setModel**, respectiv **getModel**, cu argumente specifice fiecărei componente în parte. Crearea unei clase care să reprezinte un model se va face extinzând interfaţa corespunzătoare şi implementând metodele definite de aceasta sau extinzând clasa implicită oferită de API-ul Swing şi supradefinind metodele care ne interesează. Pentru modelele mai complexe, cum ar fi cele asociate claselor `JTable`, `JTree` sau `JList` există clase abstracte care implementează interfaţa ce descrie modelul respectiv De exemplu, interfaţa model a clasei `JList` este `ListModel` care este implementată de clasele `DefaultListModel` şi `AbstractListModel`. In funcţie de necesităţi, oricare din aceste clase poate fi extinsă pentru a crea un nou model.

Listing 11.5: Folosirea mai multor modele pentru o componenta

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

class Fereastra extends JFrame implements ActionListener {
  String data1[] = {"rosu", "galben", "albastru"};
  String data2[] = {"red", "yellow", "blue"};
  int tipModel = 1;
  JList lst;
  ListModel model1, model2;

  public Fereastra(String titlu) {
    super(titlu);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Lista initiala nu are nici un model
    lst = new JList();
    getContentPane().add(lst, BorderLayout.CENTER);

    // La apasara butonului schimbam modelul
    JButton btn = new JButton("Schimba modelul");
    getContentPane().add(btn, BorderLayout.SOUTH);
    btn.addActionListener(this);

    // Cream obiectele corespunzatoare celor doua modele
    model1 = new Model1();
    model2 = new Model2();
    lst.setModel(model1);
```

```
    pack();
  }

  public void actionPerformed(ActionEvent e) {
    if (tipModel == 1) {
      lst.setModel(model2);
      tipModel = 2;
    }
    else {
      lst.setModel(model1);
      tipModel = 1;
    }
  }

  // Clasele corespunzatoare celor doua modele
  class Model1 extends AbstractListModel {
    public int getSize() {
        return data1.length;
    }
    public Object getElementAt(int index) {
      return data1[index];
    }
  }

  class Model2 extends AbstractListModel {
    public int getSize() {
        return data2.length;
    }
    public Object getElementAt(int index) {
      return data2[index];
    }
  }

}


public class TestModel {
  public static void main(String args[]) {
    new Fereastra("Test Model").show();
  }
}
```

Multe componente Swing furnizează metode care să obţină starea obiectului fără a mai fi nevoie să obţinem instanţa modelului şi să apelăm metodele

acesteia. Un exemplu este metoda `getValue` a clasei `JSlider` care este de fapt un apel de genul `getModel().getValue()`. In multe situaţii însă, mai ales pentru clase cum ar fi `JTable` sau `JTree`, folosirea modelelor aduce flexibilitate sporită programului şi este recomandată utilizarea lor.

11.5.1 Tratarea evenimentelor

Modelele componentelor trebuie să notifice apariţia unor schimbări ale datelor gestionate astfel încât să poată fi reactualizată prezentarea lor sau să fie executat un anumti cod în cadrul unui obiect de tip *listener*. In Swing, această notificare este realizată în două moduri:

1. Informativ (lightweight) - Modelele trimit un eveniment prin care sunt informaţi ascultătorii că a survenit o anumită schimbare a datelor, fără a include în eveniment detalii legate de schimbarea survenită. Obiectele de tip listener vor trebui să apeleze metode specifice componentelor pentru a afla *ce anume* s-a schimbat. Acest lucru se realizează prin intefaţa **ChangeListener** iar evenimentele sunt de tip **ChangeEvent**, modelele care suportă această abordare fiind `BoundedRangeModel`, `ButtonModel` şi `SingleSelectionModel`.

Model	**Listener**	**Tip Eveniment**
BoundedRangeModel	ChangeListener	ChangeEvent
ButtonModel	ChangeListener	ChangeEvent
SingleSelectionModelModel	ChangeListener	ChangeEvent

Interfaţa `ChangeListener` are o singură metodă:

`public void stateChanged(ChangeEvent e)`,

singura informaţie conţinută în eveniment fiind componenta sursă.

Inregistrarea şi eliminarea obiectelor de tip listener se realizează cu metodele `addChangeListener`, respectiv `removeChangeListener`.

```
JSlider slider = new JSlider();
BoundedRangeModel model = slider.getModel();
model.addChangeListener(new ChangeListener() {
   public void stateChanged(ChangeEvent e) {
      // Sursa este de tip BoundedRangeModel
      BoundedRangeModel m = (BoundedRangeModel)e.getSource();
      // Trebuie sa interogam sursa asupra schimbarii
```

```
      System.out.println("Schimbare model: " + m.getValue());
   }
});
```

Pentru uşurinţa programării, pentru a nu lucra direct cu instanţa modelului, unele clase permit înregistrarea ascultătorilor direct pentru componenta în sine, singura diferenţă faţă de varianta anterioară constând în faptul că sursa evenimentului este acum de tipul componentei şi nu de tipul modelului. Secvenţa de cod de mai sus poate fi rescrisă astfel:

```
JSlider slider = new JSlider();
slider.addChangeListener(new ChangeListener() {
   public void stateChanged(ChangeEvent e) {
      // Sursa este de tip JSlider
      JSlider s = (JSlider)e.getSource();
      System.out.println("Valoare noua: " + s.getValue());
   }
});
```

2. Consistent(statefull) - Modele pun la dispoziţie interfeţe specializate şi tipuri de evenimente specifice ce includ toate informaţiile legate de schimbarea datelor.

Model	**Listener**	**Tip Eveniment**
ListModel	ListDataListener	ListDataEvent
ListSelectionModel	ListSelectionListener	ListSelectionEvent
ComboBoxModel	ListDataListener	ListDataEvent
TreeModel	TreeModelListener	TreeModelEvent
TreeSelectionModel	TreeSelectionListener	TreeSelectionEvent
TableModel	TableModelListener	TableModelEvent
TableColumnModel	TableColumnModelListener	TableColumnModelEvent
Document	DocumentListener	DocumentEvent
Document	UndoableEditListener	UndoableEditEvent

Folosirea acestor interfeţe nu diferă cu nimic de cazul general:

```
String culori[] = {"rosu", "galben", "albastru");
JList list = new JList(culori);
ListSelectionModel sModel = list.getSelectionModel();
sModel.addListSelectionListener(
```

```
  new ListSelectionListener() {
    public void valueChanged(ListSelectionEvent e) {
      // Schimbarea este continuta in eveniment
      if (!e.getValueIsAdjusting()) {
        System.out.println("Selectie curenta: " +
          e.getFirstIndex());
      }
   }
});
```

11.6 Folosirea componentelor

Datorită complexităţii modelului Swing, în această secţtiune nu vom încerca o abordare exhaustivă a modului de utilizare a tuturor componentelor, ci vom pune în evidenţă doar aspectele specifice acestui model, subliniind diferenţele şi îmbunătăţirile faţă AWT.

11.6.1 Componente atomice

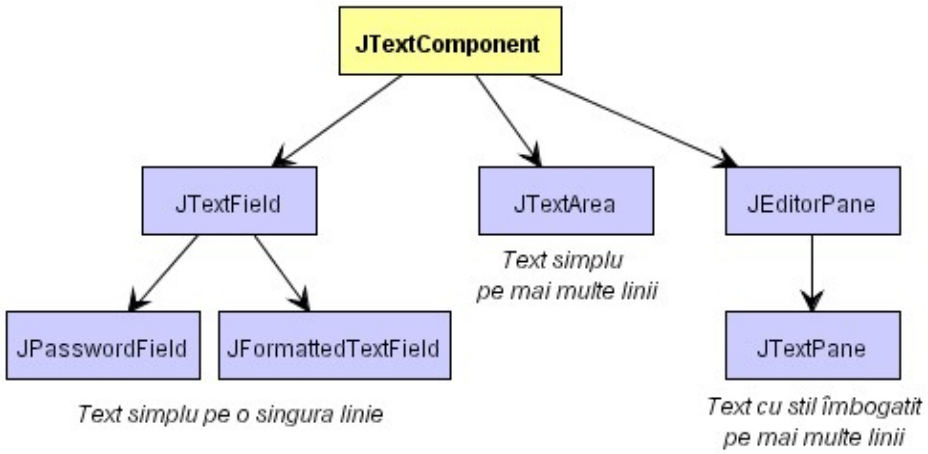
In categoria componentelor atomice includem componentele Swing cu funcţionalitate simplă, a căror folosire este în general asemănătoare cu a echivalentelor din AWT. Aici includem:

- Etichete: `JLabel`
- Butoane simple sau cu două stări: `JButton, JCheckBox, JRadioButton, JToggleButton`; mai multe butoane radio pot fi grupate folosind clasa `ButtonGroup`, pentru a permite selectarea doar a unuia dintre ele.
- Componente pentru progres şi derulare: `JSlider, JProgressBar, JScrollBar`
- Separatori: `JSeparator`

Deoarece utilizarea acestora este în general facilă, nu vom analiza în parte aceste componente.

11.6.2 Componente pentru editare de text

Componentele Swing pentru afişarea şi editarea textelor sunt grupate într-o ierarhie ce are ca rădăcină clasa **JTextComponent** din pachetul `javax.swing.text`.



După cum se observă din imaginea de mai sus, clasele pot împărţite în trei categorii, corespunzătoare tipului textului editat:

- Text simplu pe o singură linie
  - **JTextField** - Permite editarea unui text simplu, pe o singură linie.
  - **JPasswordField** - Permite editarea de parole. Textul acestora va fi ascuns, în locul caracterelor introduse fiind afişat un caracter simbolic, cum ar fi '*'.
  - **JFormattedTextField** - Permite introducerea unui text care să respecte un anumit format, fiind foarte utilă pentru citirea de numere, date calendaristice, etc. Este folosită împreună cu clase utilitare pentru formatarea textelor, cum ar fi `NumberFormatter`, `DateFormatter`, `MaskFormatter`, etc. Valoarea conţinută de o astfel de componentă va fi obţinută/setată cu metodele **getValue**, respectiv **setValue** şi nu cu cele uzuale `getText`, `setText`.
- Text simplu pe mai multe linii
  - **JTextArea** - Permite editarea unui text simplu, pe mai multe linii. Orice atribut legat de stil, cum ar fi culoarea sau fontul, se aplică întregului text şi nu poate fi specificat doar unei anumite porţiuni. Uzual, o componentă de acest tip va fi inclusă într-un container `JScrollPane`, pentru a permite navigarea pe verticală

şi orizontală dacă textul introdus nu încape în suprafaţa alocată obiectului. Acest lucru este valabil pentru toate componentele Swing pentru care are sens noţiunea de navigare pe orizontală sau verticală, nici una neoferind suport intrinsec pentru această operaţiune.

- Text cu stil îmbogăţit pe mai multe linii
  - **JEditorPane** - Permite afişarea şi editarea de texte scrise cu stiluri multiple şi care pot include imagini sau chiar diverse alet componente. Implicit, următoarele tipuri de texte sunt recunoscute: **text/plain**, **text/html** şi **text/rtf**. Una din utilizările cele mai simple ale acestei clase este setarea documentului ce va fi afişat cu metoda **setPage**, ce primeşte ca argument un URL care poate referi un fişier text, HTML sau RTF.
  - **JTextPane** - Această clasă extinde `JEditorPane`, oferind diverse facilităţi suplimentare pentru lucrul cu stiluri şi paragrafe.

Clasa `JTextComponent` încearcă să păstreze cât mai multe similitudini cu clasa `TextComponent` din AWT, însă există diferenţe notabile între cele două, componenta Swing având caracteristici mult mai complexe cum ar fi suport pentru operaţii de *undo* şi *redo*, tratarea evenimentelor generate de cursor (caret), etc. Orice obiect derivat din `JTextComponent` este format din:

- Un **model**, referit sub denumirea de *document*, care gestionează starea componentei. O referinţă la model poate fi obţinută cu metoda `getDocument`, ce returnează un obiect de tip `Document`.
- O **reprezentare**, care este responsabilă cu afişarea textului.
- Un **'controller'**, cunoscut sub numele de *editor kit* care permite scrierea şi citirea textului şi care permite definirea de acţiuni necesare editării.

Există diferenţ faţă de AWT şi la nivelul tratării evenimentelor generate de componentele pentru editarea de texte. Dintre evenimentele ce pot fi generate amintim:

- **ActionEvent** - Componentele derivate din `JTextField` vor genera un eveniment de acest tip la apăsarea tastei Enter în căsuţa de editare a textului. Interfaţa care trebuie implementată este **ActionListener**.

- **CaretEvent** - Este evenimentul generat la deplasarea cursorului ce gestionează poziţia curentă în text. Interfaţa corespunzătoare **CaretListener** conţine o singură metodă: `caretUpdate` ce va fi apelată ori de câte ori apare o schimbare.

- **DocumentEvent** - Evenimentele de acest tip sunt generate la orice schimbare a textului, sursa lor fiind documentul (modelul) componentei şi nu componenta în sine. Interfaţa corespunzătoare este **DocumentListener**, ce conţine metodele:

  - `insertUpdate` - apelată la adăugarea de noi caractere;
  - `removeUpdate` - apelată după o operaţiune de ştergere;
  - `changedUpdate` - apelată la schimbarea unor atribute legate de stilul textului.

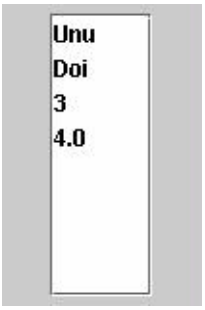
- **PropertyChangeEvent** - Este un eveniment comun tuturor componentelor de tip JavaBean, fiind generat la orice schimbare a unei proprietăţi a componentei. Interfaţa corespunzătoare este **PropertyChangeListener**, ce conţine metoda `propertyChange`.

11.6.3 Componente pentru selectarea unor elemente

In această categorie vom include clasele care permit selectarea unor valori (elemente) dintr-o serie prestabilită. Acestea sunt: **JList**, **JComboBox** şi **JSpinner**.

Clasa JList

Clasa `JList` descrie o listă de elemente dispuse pe una sau mai multe coloane, din care utilizatorul poate selecta unul sau mai multe. Uzual un obiect de acest tip va fi inclus într-un container de tip `JScrollPane`.



Iniţializarea unei liste se realizează în mai multe modalităţi:

- Folosind unul din constructorii care primesc ca argument un vector de elemente.

```
Object elemente[] = {"Unu", "Doi", new Integer(3), new Double(4)};
JList lista = new JList(elemente);
```

- Folosind constructorul fără argumente şi adăugând apoi elemente modelului implicit listei:

```
DefaultListModel model = new DefaultListModel();
model.addElement("Unu");
model.addElement("Doi");
model.addElement(new Integer(3));
model.addElement(new Double(4));
JList lista = new JList(model);
```

- Folosind un model propriu, responsabil cu furnizarea elementelor listei. Acesta este un obiect dintr-o clasă ce trebuie să implementeze interfaţa **ListModel**, uzual fiind folosită extinderea clasei predefinite **AbstractListModel** şi supradefinirea metodelor: `getElementAt` care furnizează elementul de pe o anumită poziţie din listă, respectiv `getSize` care trebuie să returneze numărul total de elemente din listă. Evident, această variantă este mai complexă, oferind flexibilitate sporită în lucrul cu liste.

```
ModelLista model = new ModelLista();
JList lista = new JList(model);
```

```
...
class ModelLista extends AbstractListModel {
  Object elemente[] = {"Unu", "Doi", new Integer(3), new Double(4)};
  public int getSize() {
   return elemente.length;
  }
  public Object getElementAt(int index) {
    return elemente[index];
  }
}
```

Gestiunea articolelor selectate dintr-o listă se realizează prin intermediul unui model, acesta fiind un obiect de tip **ListSelectionModel**. Obiectele de tip `JList` generează evenimente de tip **ListSelectionEvent**, interfaţa corespunzătoare fiind **ListSelectionListener** ce conţine metoda **valueChanged** apelată ori de câte ori va fi schimbată selecţia elementelor din listă.

```
class Test implements ListSelectionListener {
  ...
  public Test() {
    ...
   // Stabilim modul de selectie
   list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        /* sau SINGLE_INTERVAL_SELECTION
               MULTIPLE_INTERVAL_SELECTION
        */

   // Adaugam un ascultator
   ListSelectionModel model = list.getSelectionModel();
   model.addListSelectionListener(this);
    ...
  }

  public void valueChanged(ListSelectionEvent e) {
    if (e.getValueIsAdjusting()) return;
    int index = list.getSelectedIndex();
```

```
    ...
  }
}
```

Evident, clasa oferă metode pentru selectarea unor elemente din cadrul programului `setSelectedIndex, setSelectedIndices, etc.` şi pentru obţinerea celor selectate la un moment dat `getSelectedIndex, getSelectedIndices, etc.`.

O facilitate extrem de importantă pe care o au listele este posibilitatea de a stabili un *renderer* pentru fiecare articol în parte. Implicit toate elementele listei sunt afişate în acelaşi fel, însă acest lucru poate fi schimbat prin crearea unei clase ce implementează interfaţa **ListCellRenderer** şi personalizează reprezentarea elementelor listei în funcţie de diverşi parametri. Interfaţa `ListCellRenderer` conţine o singură metodă **getListCellRendererComponent** ce returnează un obiect de tip `Component`. Metoda va fi apelată în parte pentru reprezentarea fiecărui element al listei.

```
class MyCellRenderer extends JLabel implements ListCellRenderer {
  public MyCellRenderer() {
   setOpaque(true);
  }
  public Component getListCellRendererComponent(
      JList list, Object value, int index,
      boolean isSelected, boolean cellHasFocus)  {
    setText(value.toString());
    setBackground(isSelected ? Color.red : Color.white);
    setForeground(isSelected ? Color.white : Color.black);
    return this;
  }
}
```

Setarea unui anumit *renderer* pentru o listă se realizează cu metoda **setCellRenderer**.

Clasa JComboBox

Clasa `JComboBox` este similară cu `JList`, cu deosebirea că permite doar selectarea unui singur articol, acesta fiind şi singurul permanent vizibil. Lista

celorlalte elemente este afişată doar la apăsarea unui buton marcat cu o săgeată, ce face parte integrantă din componentă.

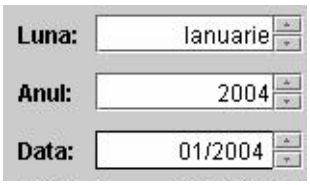
`JComboBox` funcţionează după aceleaşi principii ca şi clasa `JList`. Iniţializarea se face dintr-un vector sau folosind un model de tipul **ComboBoxModel**, fiecare element putând fi de asemenea reprezentat diferit prin intermediul unui obiect ce implementează aceeaşi intefaţă ca şi în cazul listelor: **ListCellRenderer**.

O diferenţă notabilă constă în modul de selectare a unui articol, deoarece `JComboBox` permite şi editarea explicită a valorii elementului, acest lucru fiind controlat de metoda `setEditable`.

Evenimentele generate de obiectele `JComboBox` sunt de tip `ItemEvent` generate la navigarea prin listă, respectiv `ActionEvent` generate la selectarea efectivă a unui articol.

Clasa JSpinner

Clasa `JSpinner` oferă posibilitatea de a selecta o anumită valoare (element) dintr-un domeniu prestabilit, lista elementelor nefiind însă vizibilă. Este folosit atunci când domeniul din care poate fi făcută selecţia este foarte mare sau chiar nemărginit; de exemplu: numere intregi intre 1950 si 2050. Componenta conţine două butoane cu care poate fi selectat următorul, respectiv predecesorul element din domeniu.



`JSpiner` se bazează exclusiv pe folosirea unui model. Acesta este un obiect de tip **SpinnerModel**, existând o serie de clase predefinite ce implementează această interfaţă cum ar fi `SpinnerListModel`, `SpinnerNumberModel` sau `SpinnerDateModel` ce pot fi utilizate.

Componentele de acest tip permit şi specificarea unui anumit tip de editor pentru valorile elementelor sale. Acesta este instalat automat pentru fiecare din modelele standard amintite mai sus, fiind reprezentat de una din clasele `JSpinner.ListEditor`, `JSpinner.NumberEditor`, respectiv `JSpinner.DateEditor`, toate derivate din `JSpinner.DefaultEditor`. Fiecare din editoarele amintite permite diverse formatări specifice.

Evenimentele generate de obiectele de tip `JSpinner` sunt de tip `ChangeEvent`, generate la schimbarea stării componentei.

11.6.4 Tabele

Clasa **JTable** permite crearea de componente care să afişeze o serie de elemente într-un format tabelar, articolele fiind dispuse pe linii şi coloane. Un tabel poate fi folosit doar pentru afişarea formatată a unor date, dar este posibilă şi editarea informaţiei din celulele sale. De asemenea, liniile tabelului pot fi marcate ca selectate, tipul selecţiei fiind simplu sau compus, tabelele extinzând astfel funcţionalitatea listelor.

Nume	Varsta	Student
Ionescu	20	true
Popescu	80	false

Deşi clasa `JTable` se găseşte în pachetul `javax.swing`, o serie de clase şi interfeţe necesare lucrului cu tabele se găsesc în pachetul **javax.swing.table**, acesta trebuind aşadar importat.

Iniţializarea unui tabel poate fi făcută în mai multe moduri.
Cea mai simplă variantă este să folosim unul din constructorii care primesc ca argumente elementele tabelului sub forma unei matrici sau a unei colecţii de tip `Vector` şi denumirile capurilor de coloană:

```
String[] coloane = {"Nume", "Varsta", "Student"};
Object[][] elemente = {
    {"Ionescu", new Integer(20), Boolean.TRUE},
    {"Popescu", new Integer(80), Boolean.FALSE}};
JTable tabel = new JTable(elemente, coloane);
```

După cum se observă, tipul de date al elementelor de pe o coloană este de tip referinţă şi poate fi oricare. In cazul în care celulele tabelului sunt

editabile trebuie să existe un editor potrivit pentru tipul elementului din celula respectivă. Din motive de eficienţă, implementarea acestei clase este orientată la nivel de coloană, ceea ce înseamnă că articole de pe o coloană vor fi reprezentate la fel şi vor avea acelaşi tip de editor.

A doua variantă de creare a unui tabel este prin implementarea modelului acestuia într-o clasă separată şi folosirea constructorului corespunzător. Interfaţa care descrie modelul clasei `JTable` este **TableModel** şi conţine metodele care vor fi interogate pentru obţinerea informaţiei din tabel. Uzual, crearea unui model se face prin extinderea clasei predefinite **AbstractTableModel**, care implementează deja `TableModel`. Tot ceea ce trebuie să facem este să supradefinim metodele care ne interesează, cele mai utilizate fiind (primele trei trebuie obligatoriu supradefinite, ele fiind declarate abstracte în clasa de bază):

- **getRowCount** - returnează numărul de linii ale tabelului;
- **getColumnCount** - returnează numărul de coloane ale tabelului;
- **getValueAt** - returnează elementul de la o anumită linie şi coloană;
- **getColumnName** - returnează denumirea fiecărei coloane;
- **isCellEditable** - specifică dacă o anumită celulă este editabilă.

Modelul mai conţine şi metoda **setValueAt** care poate fi folosită pentru setarea explicită a valorii unei celule.

```
ModelTabel model = new ModelTabel();
JTable tabel = new JTable(model);
...
class ModelTabel extends AbstractTableModel {
  String[] coloane = {"Nume", "Varsta", "Student"};
  Object[][] elemente = {
    {"Ionescu", new Integer(20), Boolean.TRUE},
    {"Popescu", new Integer(80), Boolean.FALSE}};

  public int getColumnCount() {
    return coloane.length;
  }
  public int getRowCount() {
```

```
    return elemente.length;
  }
  public Object getValueAt(int row, int col) {
    return elemente[row][col];
  }
  public String getColumnName(int col) {
    return coloane[col];
  }
  public boolean isCellEditable(int row, int col) {
    // Doar numele este editabil
    return (col == 0);
  }
}
```

Orice schimbare a datelor tabelului va genera un eveniment de tip **TableModelEvent**. Pentru a trata aceste evenimente va trebui să implementăm interfaţa **TableModelListener** ce conţine metoda **tableChanged**. Inregistrarea unui listener va fi făcută pentru modelul tabelului:

```
public class Test implements TableModelListener {
  ...
  public Test() {
    ...
    tabel.getModel().addTableModelListener(this);
    ...
  }
  public void tableChanged(TableModelEvent e) {
    // Aflam celula care a fost modificata
    int row = e.getFirstRow();
    int col = e.getColumn();
    TableModel model = (TableModel)e.getSource();
    Object data = model.getValueAt(row, col);
    ...
  }
}
```

Tabele oferă posibilitatea de a selecta una sau mai multe linii, nu neapărat consecutive, gestiunea liniilor selectate fiind realizată prin intermediul unui model. Acesta este o instanţă ce implementează, întocmai ca la liste, interfaţa **ListSelectionModel**. Tratarea evenimentelor generate de schimbarea selecţiei în tabel se realizează prin înregistrarea unui ascultător de tip **ListSelectionListener**:

```
class Test implements ListSelectionListener {
  ...
  public Test() {
    ...
    // Stabilim modul de selectie
   tabel.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

    // Adaugam un ascultator
   ListSelectionModel model = tabel.getSelectionModel();
   model.addListSelectionListener(this);
    ...
  }

  public void valueChanged(ListSelectionEvent e) {
    if (e.getValueIsAdjusting()) return;
    ListSelectionModel model =
      (ListSelectionModel)e.getSource();
    if (model.isSelectionEmpty()) {
      // Nu este nici o linie selectata
      ...
    } else {
      int index = model.getMinSelectionIndex();
      // Linia cu numarul index este prima selectata
      ...
    }
  }
}
```

După cum am spus, celule unei coloane vor fi reprezentare la fel, fiecare coloană având asociat un obiect *renderer* responsabil cu crearea componen-

tei ce descrie celulele sale. Un astfel de obiect implementează interfaţa **TableCellRenderer**, care are o singură metodă **getTableCellRendererComponent**, aceasta fiind responsabilă cu crearea componentelor ce vor fi afişate în celulele unei coloane. Implicit, există o serie de tipuri de date cu reprezentări specifice, cum ar fi: `Boolean, Number, Double, Float, Date, ImageIcon, Icon`, restul tipurilor având o reprezentare standard ce constă într-o etichetă cu reprezentarea obiectului ca şir de caractere. Specificarea unui *renderer* propriu se realizează cu metoda **setDefaultRenderer**, ce asociază un anumit tip de date cu un obiect de tip `TableRenderer`.

```
public class MyRenderer extends JLabel
    implements TableCellRenderer {
  public Component getTableCellRendererComponent(...) {
    ...
    return this;
  }
}
```

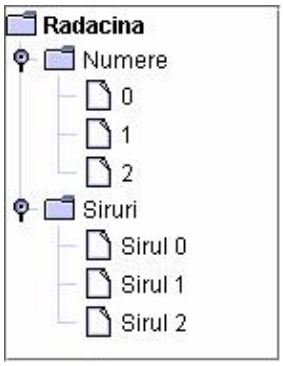
O situaţie similară o regăsim la nivelul editorului asociat celulelor dintr-o anumită coloană. Acesta este un obiect ce implementează interfaţa **TreeCellEditor**, ce extinde interfaţa **CellEditor** care generalizează conceptul de celulă editabilă pe care îl vom mai regăsi la arbori. Implicit, există o serie de editoare standard pentru tipurile de date menţionate anterior, dar este posibilă specificarea unui editor propriu cu metoda **setDefaultEditor**. Crearea unui editor propriu se realizează cel mai simplu prin extinderea clasei utilitare **AbstractCellEditor**, care implementează `CellEditor`, plus implementarea metodei specifice din `TreeCellEditor`.

```
public class MyEditor
    extends AbstractCellEditor
    implements TableCellEditor {

  // Singura metoda abstracta a parintelui
  public Object getCellEditorValue() {
    // Returneaza valoarea editata
    ...
  }
```

```
  // Metoda definita de TableCellEditor
  public Component getTableCellEditorComponent(...) {
    // Returneaza componenta de tip editor
    ...
  }
}
```

11.6.5 Arbori

Clasa **JTree** permite afişarea unor elemente într-o manieră ierarhică. Ca orice componentă Swing netrivială, un obiect `JTree` reprezintă doar o imagine a datelor, informaţia în sine fiind manipulată prin intermediul unui model. La nivel structural, un arbore este format dintr-o *rădăcină*, *noduri interne* - care au cel puţin un fiu şi *noduri frunză* - care nu mai au nici un descendent.



Deşi clasa `JTree` se găseşte în pachetul `javax.swing`, o serie de clase şi interfeţe necesare lucrului cu arbori se găsesc în pachetul **javax.swing.tree**.

Clasa care modelează noţiunea de nod al arborelui este **DefaultMutableTreeNode**, aceasta fiind folosită pentru toate tipurile de noduri. Crearea unui arbore presupune aşadar crearea unui nod (rădăcina), instanţierea unui obiect de tip `JTree` cu rădăcina creată şi adăugarea apoi de noduri frunză ca fii ai unor noduri existente.

```
  String text = "<html><b>Radacina</b></html>";
  DefaultMutableTreeNode root = new DefaultMutableTreeNode(text);
  DefaultMutableTreeNode numere =
      new DefaultMutableTreeNode("Numere");
  DefaultMutableTreeNode siruri =
```

```
      new DefaultMutableTreeNode("Siruri");

  for(int i=0; i<3; i++) {
    numere.add(new DefaultMutableTreeNode(new Integer(i)));
    siruri.add(new DefaultMutableTreeNode("Sirul " + i));
  }

  root.add(numere);
  root.add(siruri);

  JTree tree = new JTree(root);
```

După cum se observă, nodurile arborelui pot fi de tipuri diferite, reprezentarea lor implicită fiind obţinutăprin apelarea metodei `toString` pentru obiectului conţinut. De asemenea, este posibilă specificarea unui text în format HTML ca valoare a unui nod, acesta fiind reprezentat ca atare.

Dacă varianta adăugării explicite a nodurilor nu este potrivită, se poate implementa o clasă care să descrie modelul arborelui. Aceasta trebuie să implementeze intefaţa **TreeModel**.

Scopul unei componente de tip arbore este în general selectarea unui nod al ierarhiei. Ca şi în cazul listelor sau a tabelelor, gestiunea elementelor selectate se realizează printr-un model, în această situaţie interfaţa corespunzătoare fiind **TreeSelectionModel**. Arborii permit înregistrarea unor obiecte *listener*, de tip **TreeSelectionListener**, care să trateze evenimentele generate la schimbarea selecţiei în arbore.

```
class Test implements TreeSelectionListener {
  ...
  public Test() {
    ...
    // Stabilim modul de selectie
    tree.getSelectionModel().setSelectionMode(
      TreeSelectionModel.SINGLE_TREE_SELECTION);

    // Adaugam un ascultator
    tree.addTreeSelectionListener(this);
    ...
  }
```

```
  public void valueChanged(TreeSelectionEvent e) {
    // Obtinem nodul selectat
    DefaultMutableTreeNode node = (DefaultMutableTreeNode)
           tree.getLastSelectedPathComponent();

    if (node == null) return;

    // Obtinem informatia din nod
    Object nodeInfo = node.getUserObject();
    ...
    }
}
```

Fiecare nod al arborelui este reprezentar prin intermediul unei clase *renderer*. Aceasta implementează interfaţa **TreeCellRenderer**, cea folosită implicit fiind **DefaultTreeCellRenderer**. Prin implementarea interfeţei sau extinderea clasei implicite pot fi create modalităţi de personalizare a nodurilor arborelui în funcţie de tipul sau valoarea acestora.

Există însă şi diverse metode de a schimba înfăţişarea unui arbore fără să creăm noi clase de tip `TreeCellRenderer`. Acestea sunt:

- `setRootVisible` - Specifică dacă rădăcina e vizibilă sau nu;
- `setShowsRootHandles` - Specifică dacă nodurile de pe primul nivel au simboluri care să permită expandarea sau restrângerea lor.
- `putClientProperty` - Stabileşte diverse proprietăţi, cum ar fi modul de reprezentare a relaţiilor (liniilor) dintre nodurile părinte şi fiu:

  ```
  tree.putClientProperty("JTree.lineStyle", "Angled");
  // sau "Horizontal", "None"
  ```

- Specificarea unei iconiţe pentru nodurile frunză sau interne:

  ```
  ImageIcon leaf = createImageIcon("img/leaf.gif");
  ImageIcon open = createImageIcon("img/open.gif");
  ImageIcon closed = createImageIcon("img/closed.gif");
  ```

```
DefaultTreeCellRenderer renderer =
  new DefaultTreeCellRenderer();
renderer.setLeafIcon(leaf);
renderer.setOpenIcon(open);
renderer.setClosedIcon(closed);

tree.setCellRenderer(renderer);
```

11.6.6 Containere

După cum ştim, containerele reprezintă suprafeţ de afişare pe care pot fi plasate ale componente, eventual chiar alte containere. Superclasa componentelor de acest tip este **Container**, clasă despre care am mai discutat în capitolul dedicat modeluli AWT.

Containerele pot fi împoărţite în două categorii:

1. **Containere de nivel înalt** - Acestea sunt `JFrame`, `JDialog`, `JApplet` şi reprezintă rădăcinile ierarhiilor de componente ale unei aplicaţii.
2. **Containere intermediare** - Reprezintă suprafeţe de afişare cu ajutorul cărora pot fi organizate mai eficient componentele aplicaţiei, putând fi imbricate. Cele mai importante clase care descriu astfel de containere sunt:
   - `JPanel`
   - `JScrollPane`
   - `JTabbedPane`
   - `JSplitPane`
   - `JLayeredPane`
   - `JDesktopPane`
   - `JRootPane`

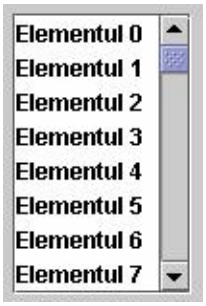
JPanel are aceeaşi funcţionalitate ca şi clasa `Panel` din AWT, fiind folosit pentru gruparea mai multor componente Swing şi plasarea lor împreună pe o altă suprafaţă de afişare. Gestionarul de poziţionare implicit este `FlowLayout`, acesta putând fi schimbat însă chiar în momentul construirii

obiectului `JPanel` sau ulterior cu metoda `setLayout`. Adăugarea de componente se realizează ca pentru orice container, folosind metoda `add`.

```
JPanel p = new JPanel(new BorderLayout());
/* Preferabil, deoarece nu mai este construit si
   un obiect de tip FlowLayout (implicit)
*/
p.add(new JLabel("Hello"));
p.add(new JButton("OK"));
...
```

JScrollPane este o clasă foarte importantă în arhitectura modelului Swing, deoarece oferă suport pentru derularea pe orizontală şi verticală a componentelor a căror reprezentare completă nu încape în suprafaţa asociată, nici o componentă Swing neoferind suport intrinsec pentru această operaţie.

```
String elemente[] = new String[100];
for(int i=0; i<100; i++)
 elemente[i] = "Elementul " + i;

JList lista = new JList(elemente);
JScrollPane sp = new JScrollPane(lista);
frame.getContentPane().add(sp);
```



JTabbedPane este utilă pentru suprapunerea mai multor containere, uzual panouri (obiecte de tip `JPanel`), pe acelaşi spaţiu de afişare, selectarea

unuia sau altui panou realizându-se prin intermediul unor butoane dispuse pe partea superioară a componentei, fiecare panou având un astfel de buton corespunzător. Ca funcţionalitate, oferă o implementare asemănătoare gestionarului de poziţionare `CardLayout`.

```
JTabbedPane tabbedPane = new JTabbedPane();
ImageIcon icon = new ImageIcon("smiley.gif");

JComponent panel1 = new JPanel();
panel1.setOpaque(true);
panel1.add(new JLabel("Hello"));
tabbedPane.addTab("Tab 1", icon, panel1,
        "Aici avem o eticheta");
tabbedPane.setMnemonicAt(0, KeyEvent.VK_1);

JComponent panel2 = new JPanel();
panel2.setOpaque(true);
panel2.add(new JButton("OK"));
tabbedPane.addTab("Tab 2", icon, panel2,
        "Aici avem un buton");
tabbedPane.setMnemonicAt(1, KeyEvent.VK_2);
```

JSplitPane permite crearea unui container care conţine două componente dispuse fie una lângă cealaltă, fie una sub alta şi separarea acestora prin intermediul unei bare care să permită configurarea suprafeţei alocate fiecărei componente.

```
String elem[] = {"Unu", "Doi", "Trei" };
JList list = new JList(elem);
```

```
JPanel panel = new JPanel(new GridLayout(3, 1));
panel.add(new JButton("Adauga"));
panel.add(new JButton("Sterge"));
panel.add(new JButton("Salveaza"));

JTextArea text = new JTextArea(
    "Mai multe componente separate prin\n" +
    "intermediul containerelor JSplitPane");

// Separam lista de grupul celor trei butoane
JSplitPane sp1 = new JSplitPane(
    JSplitPane.HORIZONTAL_SPLIT, list, panel);

// Separam containerul cu lista si butoanele
// de componenta pentru editare de text
JSplitPane sp2 = new JSplitPane(
    JSplitPane.VERTICAL_SPLIT, sp1, text);

frame.getContentPane().add(sp2);
```

11.6.7 Dialoguri

Clasa care descrie ferestre de dialog este `JDialog`, crearea unui dialog realizându-se prin extinderea acesteia, întocmai ca în modelul AWT. In Swing există însă o serie de clase predefinite ce descriu anumite tipuri de dialoguri, extrem de utile în majoritatea aplicaţiilor. Acestea sunt:

- **JOptionPane** - Permite crearea unor dialoguri simple, folosite pentru afişarea unor mesaje, realizarea unor interogări de confirmare/renunţare,

etc. sau chiar pentru introducerea unor valori, clasa fiind extrem de configurabilă. Mai jos, sunt exemplificate două modalităţi de utilizare a clasei:

```
JOptionPane.showMessageDialog(frame,
    "Eroare de sistem !",  "Eroare",
    JOptionPane.ERROR_MESSAGE);

JOptionPane.showConfirmDialog(frame,
    "Doriti inchiderea aplicatiei ? ", "Intrebare",
    JOptionPane.YES_NO_OPTION,
    JOptionPane.QUESTION_MESSAGE);
```

- **JFileChooser** - Dialog standard care permite navigarea prin sistemul de fişiere şi selectarea unui anumit fişier pentru operaţii de deschidere, respectiv salvare.
- **JColorChooser** - Dialog standard pentru selectarea într-o manieră facilă a unei culori.
- **ProgressMonitor** - Clasă utilizată pentru monitorizareă progresului unei operaţii consumatoare de timp.

11.7 Desenarea

11.7.1 Metode specifice

După cum ştim, desenarea unei componente este un proces care se executa automat ori de câte ori este necesar. Procesul în sine este asemănător celui din modelul AWT, însă există unele diferenţe care trebuie menţionate.

Orice componentă se găseşte într-o ierarhie formată de containere, rădăcina acestei fiind un container de nivel înalt, cum ar fi o fereastră sau suprafaţa unui applet. Cu alte cuvinte, componenta este plasată pe o suprafaţă de afişare, care la rândul ei poate fi plasată pe altă suprafaţă şi aşa mai departe. Când este necesară desenarea componentei repsective, fie la prima sa afişare, fie ca urmare a unor acţiuni externe sau interne programului, operaţia de desenare va fi executată pentru toate containerele, începând cu cel de la nivelul superior.

Desenarea se bazează pe modelul AWT, metoda cea mai importantă fiind **paint**, apelată automat ori de câte ori este necesar. Pentru componentele Swing, această metodă are însă o implementare specifică **şi nu trebuie supradefinită**. Aceasta este responsabilă cu apelul metodelor Swing ce desenează componenta şi anume:

- **paintComponent** - Este principala metodă pentru desenare ce este supradefinită pentru fiecare componentă Swing în parte pentru a descrie reprezentarea sa grafică. Implicit, în cazul în care componenta este opacă metoda desenează suprafaţa sa cu culoarea de fundal, după care va executa desenarea propriu-zisă.

- **paintBorder** - Desenează chenarele componentei (dacă există). Nu trebuie supradefinită.

- **paintChildren** - Solicită desenarea componentelor conţinute de această componentă (dacă există). Nu trebuie supradefinită.

Metoda `paint` este responsabilă cu apelul metodelor amintite mai sus şi realizarea unor optimizări legate de procesul de desenare, cum ar fi implementarea mecanismului de *double-buffering*. Deşi este posibilă supradefinirea ei, acest lucru nu este recomandat, din motivele amintite mai sus.

Ca şi în AWT, dacă se doreşte redesenarea explicită a unei componente se va apela metoda **repaint**. In cazul în care dimensiunea sau poziţia componentei s-au schimbat, apelul metodei **revalidate** va precede apelul lui `repaint`.

Atenţie

Intocmai ca în AWT, desenarea este realizată de firul de execuţie care se ocupă cu transmiterea evenimentelor. Pe perioada în care acesta este ocupat cu transmiterea unui mesaj nu va fi făcută nici o desenare. De asemenea, dacă acesta este blocat într-o operaţiune de desenare ce consumă mult timp, pe perioada respectivă nu va fi transmis nici un mesaj.

11.7.2 Consideraţii generale

In continuare vom prezenta câteva consideraţii generale legate de diferite aspecte ale desenării în cadrul modelului Swing.

Afişarea imaginilor

In AWT afişarea unei imagini era realizată uzual prin supradefinirea clasei `Canvas` şi desenarea imaginii în metoda `paint` a acesteia. In Swing, există câteva soluţii mai simple pentru afişarea unei imagini, cea mai utilizată fiind crearea unei etichete (`JLabel`) sau a unui buton (`JButton`) care să aibă setată o anumită imagine pe suprafaţa sa. Imaginea respectivă trebuie creată folosind clasa **ImageIcon**.

```
ImageIcon img = new ImageIcon("smiley.gif");
JLabel label = new JLabel(img);
```

Transparenţa

Cu ajutorul metodei `setOpaque` poate fi controlată opacitatea componentelor Swing. Aceasta este o facilitate extrem de importantă deoarece permite crearea de componente care nu au formă rectangulară. De exemplu, un buton circular va fi construit ca fiind transparent (`setOpaque(false)`) şi va desena în interiorul său o elipsă umplută cu o anumită culoare. Evident, este necesară implementarea de cod specific pentru a trata apăsarea acestui tip de buton.

Trabsparenţa însă vine cu un anumit preţ, deoarece pentru componentele transparente vor trebui redesenate containerele pe care se găseşte aceasta, încetinind astfel procesul de afişare. Din acest motiv, de fiecare dată când este cazul, se recomandă setarea componentelor ca fiind opace (`setOpaque(true)`).

Dimensiunile componentelor

După cum ştim, orice componentă este definită de o suprafaţă rectangulară. Dimensiunile acestei pot fi obţinute cu metodele `getSize, getWidth, getHeight`. Acestea includ însă şi dimsniunile chenarelor, evident dacă acestea există. Suprafaţa ocupată de acestea poate fi aflată cu metoda **getInsets**

ce va returna un obiect de tip **Insets** ce specifică numărul de pixeli ocupaţi cu chenare în jurul componentei.

```
public void paintComponent(Graphics g) {
    ...
    Insets insets = getInsets();
    int currentWidth = getWidth() - insets.left - insets.right;
    int currentHeight = getHeight() - insets.top - insets.bottom;
    ...
}
```

Contexte grafice

Argumentul metodei `paintComponent` este de tip **Graphics** ce oferă primitivele standard de desenare. In majoritatea cazurilor însă, argumentul este de fapt de tip **Graphics2D**, clasă ce extinde `Graphics` şi pune la dispoziţie metode mai sofisitcate de desenare cunoscute sub numele de Java2D. Pentru a avea acces la API-ul Java2D, este suficient să facem conversia argumentului ce descrie contextul grafic:

```
public void paintComponent(Graphics g) {
  Graphics2D g2d = (Graphics2D)g;
  // Desenam apoi cu g2d
  ...
}
```

In Swing, pentru a eficientiza desenarea, obiectul de tip `Graphics` primit ca argument de metoda `paintComponent` este refolosit pentru desenarea componentei, a chenarelor şi a fiilor săi. Din acest motiv este foarte important ca atunci când supradefinim metoda `paintComponent` să ne asigurăm că la terminarea metodei starea obiectului `Graphics` este aceeaşi ca la început. Acest lucru poate fi realizat fie explicit, fie folosind o copie a contextului grafic primit ca argument:

```
// 1.Explicit
Graphics2D g2d = (Graphics2D)g;
g2d.translate(x, y);   // modificam contexul
...
g2d.translate(-x, -y); // revenim la starea initiala
```

```
// 2. Folosirea unei copii
Graphics2D g2d = (Graphics2D)g.create();
g2d.translate(x, y);
...
g2d.dispose();
```

11.8 Look and Feel

Prin sintagma *'Look and Feel' (L&F)* vom înţelege modul în care sunt desenate componentele Swing şi felul în care acestea interacţionează cu utilizatorul. Posibilitatea de a alege între diferite moduri L&F are avantajul de a oferi prezentarea unei aplicaţii într-o formă grafică care să corespundă preferinţelor utilizatorilor. In principiu, variantele originale de L&F furnizate în distribuţia standard ofereau modalitatea ca o interfaţă Swing fie să se încadreze în ansamblul grafic al sistemului de operare folosit, fie să aibă un aspect specific Java.

Orice L&F este descris de o clasă derivată din **LookAndFeel**. Distribuţia standard Java include următoarele clase ce pot fi utilizate pentru selectarea unui L&F:

- `javax.swing.plaf.metal.MetalLookAndFeel`
  Este varianta implicită de L&F şi are un aspect specific Java.

- `com.sun.java.swing.plaf.windows.WindowsLookAndFeel`
  Varianta specifică sistemelor de operare Windows. Incepând cu versiunea 1.4.2 există şi implementarea pentru Windows XP .

- `com.sun.java.swing.plaf.mac.MacLookAndFeel`
  Varianta specifică sistemelor de operare Mac.

- `com.sun.java.swing.plaf.motif.MotifLookAndFeel`
  Specifică interfaţa CDE/Motif.

- `com.sun.java.swing.plaf.gtk.GTKLookAndFeel`
  GTK+ reprezintă un standard de creare a interfeţelor grafice dezvoltat independent de limbajul Java. (GTK este acronimul de la *GNU Image Manipulation Program Toolkit*). Folosind acest L&F este posibilă şi

specificarea unei anumite teme prin intermediul unui fişier de resurse sau folosind variabila `swing.gtkthemefile`, ca în exemplul de mai jos:

```
java -Dswing.gtkthemefile=temaSpecifica/gtkrc App
```

Specificare unei anumite interfeţe L&F poate fi realizată prin mai multe modalităţi.

Folosirea clasei UImanager

Clasa `UIManager` pune la dispoziţie o serie de metode statice pentru selectarea la momentul execuţiei a uni anumit L&F, precum şi pentru obţinerea unor variante specifice:

- `getLookAndFeel` - Obţine varianta curentă, returnând un obiect de tip `LookAndFeel`.
- `setLookAndFeel` - Setează modul curet L&F. Metoda primeşte ca argument un obiect dintr-o clasă derivată din `LookAndFeel`, fie un şir de caractere cu numele complet al clasei L&F.
- `getSystemLookAndFeelClassName` - Obţine variantă specifică sistemului de operare folosit. In cazul în care nu există nici o astfel de clasă, returnează varianta standard.
- `getCrossPlatformLookAndFeelClassName` - Returnează interfaţa grafică standard Java (JLF).

```
// Exemple:
UIManager.setLookAndFeel(
   "com.sun.java.swing.plaf.motif.MotifLookAndFeel");

UIManager.setLookAndFeel(
   UIManager.getSystemLookAndFeelClassName());
```

Setarea proprietăţii swing.defaultlaf

Există posibilitatea de a specifica varianta de L&F a aplicaţie direct de la linia de comandă prin setarea proprietâţii **swing.defaultlaf**:

```
java -Dswing.defaultlaf=
      com.sun.java.swing.plaf.gtk.GTKLookAndFeel App
java -Dswing.defaultlaf=
      com.sun.java.swing.plaf.windows.WindowsLookAndFeel App
```

O altă variantă de a seta această proprietate este schimbarea ei direct în fişierul `swing.properties` situat în subdirectorul `lib` al distribuţiei Java.

```
# Swing properties
swing.defaultlaf=
      com.sun.java.swing.plaf.windows.WindowsLookAndFeel
```

Ordinea în care este aleasă clasa L&F este următoarea:

1. Apelul explicit al metodei `UIManager.setLookAndFeel` înaintea creării unei componente Swing.
2. Proprietatea `swing.defaultlaf` specificată de la linia de comandă.
3. Proprietatea `swing.defaultlaf` specificată în fişierul `swing.properties`.
4. Clasa standard Java (JLF).

Există posibilitatea de a schimba varianta de L&F chiar şi după afişarea componentelor. Acesta este un proces care trebuie să actualizeze ierarhiile de componente, începând cu containerele de nivel înalt şi va fi realizat prin apelul metodei **SwingUtilities.updateComponentTreeUI** ce va primi ca argument rădăcina unei ierarhii. Secvenţa care efectuează această operaţie pentru o fereastră *f* este:

```
UIManager.setLookAndFeel(numeClasaLF);
SwingUtilities.updateComponentTreeUI(f);
f.pack();
```

Capitolul 12

Fire de execuţie

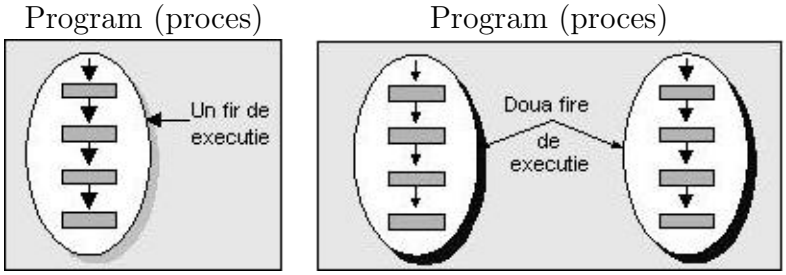
12.1 Introducere

Firele de execuţie fac trecerea de la programarea secvenţială la programarea concurentă. Un program secvenţial reprezintă modelul clasic de program: are un început, o secvenţă de execuţie a instrucţiunilor sale şi un sfârşit. Cu alte cuvinte, la un moment dat programul are un singur punct de execuţie. Un program aflat în execuţie se numeşte *proces*. Un sistem de operare monotasking, cum ar fi fi MS-DOS, nu este capabil să execute decât un singur proces la un moment dat, în timp ce un sistem de operare multitasking, cum ar fi UNIX sau Windows, poate rula oricâte procese în acelaşi timp (concurent), folosind diverse strategii de alocare a procesorului fiecăruia dintre acestea. Am reamintit acest lucru deoarece noţiunea de fir de execuţie nu are sens decât în cadrul unui sistem de operare multitasking.

Un fir de execuţie este similar unui proces secvenţial, în sensul că are un început, o secvenţă de execuţie şi un sfârşit. Diferenta dintre un fir de execuţie şi un proces constă în faptul că un fir de execuţie nu poate rula independent ci trebuie să ruleze în cadrul unui proces.

Definiţie

Un *fir de execuţie* este o succesiune scevenţială de instrucţiuni care se execută în cadrul unui proces.



Un program îşi poate defini însă nu doar un fir de execuţie ci oricâte, ceea ce înseamnă că în cadrul unui proces se pot executa simultan mai multe fire de execuţie, permiţând efectuarea concurentă a sarcinilor independente ale acelui program.

Un fir de execuţie poate fi asemănat cu o versiune redusă a unui proces, ambele rulând simultan şi independent pe o structură secvenţială formată de instrucţiunile lor. De asemenea, execuţia simultană a firelor în cadrul unui proces este similară cu execuţia concurentă a proceselor: sistemul de operare va aloca procesorul după o anumită strategie fiecărui fir de execuţie până la terminarea lor. Din acest motiv firele de execuţie mai sunt numite şi *procese usoare*.

Care ar fi însă deosebirile între un fir de execuţie şi un proces ? In primul, rând deosebirea majoră constă în faptul că firele de execuţie nu pot rula decât în cadrul unui proces. O altă deosebire rezultă din faptul că fiecare proces are propria sa memorie (propriul său spaţiu de adrese) iar la crearea unui nou proces (fork) este realizată o copie exactă a procesului părinte: cod şi date, în timp ce la crearea unui fir nu este copiat decât codul procesului părinte, toate firele de execuţie având acces la aceleaşi date, datele procesului original. Aşadar, un fir mai poate fi privit şi ca un *context de execuţie* în cadrul unui proces.

Firele de execuţie sunt utile în multe privinţe, însă uzual ele sunt folosite pentru executarea unor operaţii consumatoare de timp fără a bloca procesul principal: calcule matematice, aşteptarea eliberării unei resurse, desenarea componentelor unei aplicaţii GUI, etc. De multe ori ori, firele îşi desfăşoară activitatea în fundal însă, evident, acest lucru nu este obligatoriu.

12.2 Crearea unui fir de execuţie

Ca orice alt obiect Java, un fir de execuţie este o instanţă a unei clase. Firele de execuţie definite de o clasă vor avea acelaşi cod şi, prin urmare, aceeaşi

secvenţa de instrucţiuni. Crearea unei clase care să definească fire de execuţie poate fi facută prin două modalităţi:

- prin extinderea clasei `Thread`.
- prin implementarea interfeţei `Runnable`.

Orice clasă ale cărei instanţe vor fi executate separat într-un fir propriu trebuie declarată ca fiind de tip **Runnable**. Aceasta este o interfaţă care conţine o singură metodă şi anume metoda **run**. Aşadar, orice clasă ce descrie fire de execuţie va conţine metoda `run` în care este implementat codul ce va fi rulat. Interfaţa `Runnable` este concepută ca fiind un protocol comun pentru obiectele care doresc să execute un anumit cod pe durata existenţei lor.

Cea mai importantă clasă care implementează interfaţa `Runnable` este **Thread**. Aceasta implementează un fir de execuţie generic care, implicit, nu face nimic; cu alte cuvinte, metoda `run` nu conţine nici un cod. Orice fir de execuţie este o instanţă a clasei `Thread` sau a unei subclase a sa.

12.2.1 Extinderea clasei Thread

Cea mai simplă metodă de a crea un fir de execuţie care să realizeze o anumită acţiune este prin extinderea clasei `Thread` şi supradefinirea metodei `run` a acesteia. Formatul general al unei astfel de clase este:

```
public class FirExcecutie extends Thread {

  public FirExcecutie(String nume) {
    // Apelam constructorul superclasei
    super(nume);
  }

  public void run() {
    // Codul firului de executie
    ...
  }
}
```

Prima metodă a clasei este constructorul, care primeşte ca argument un şir ce va reprezenta numele firului de execuţie. In cazul în care nu vrem să dăm nume firelor pe care le creăm, atunci putem renunţa la supradefinirea

acestui constructor şi să folosim constructorul implicit, fără argumente, care creează un fir de execuţie fără nici un nume. Ulterior, acesta poate primi un nume cu metoda `setName`. Evident, se pot defini şi alţi constructori, aceştia fiinde utili atunci când vrem să trimitem diverşi parametri de iniţializare firului nostru. A doua metodă este metoda `run`, "inima" oricărui fir de execuţie, în care scriem efectiv codul care trebuie să se execute.

Un fir de execuţie creat nu este automat pornit, lansarea să fiind realizează de metoda **start**, definită în clasa `Thread`.

```
// Cream firul de executie
FirExecutie fir = new FirExecutie("simplu");

// Lansam in executie
fir.start();
```

Să considerăm în continuare un exemplu în care definim un fir de execuţie ce afişează numerele întregi dintr-un interval, cu un anumit pas.

Listing 12.1: Folosirea clasei `Thread`

```
class AfisareNumere extends Thread {
  private int a, b, pas;

  public AfisareNumere(int a, int b, int pas) {
    this.a = a;
    this.b = b;
    this.pas = pas;
  }

  public void run() {
    for(int i = a; i <= b; i += pas)
      System.out.print(i + " " );
  }
}

public class TestThread {
  public static void main(String args[]) {
  AfisareNumere fir1, fir2;

  fir1 = new AfisareNumere(0, 100, 5);
  // Numara de la 0 la 100 cu pasul 5

  fir2 = new AfisareNumere(100, 200, 10);
```

```
  // Numara de la 100 la 200 cu pasul 10

  fir1.start();
  fir2.start();
  // Pornim firele de executie
  // Ele vor fi distruse automat la terminarea lor
  }
}
```

Gândind secvenţial, s-ar crede că acest program va afişa prima dată numerele de la 0 la 100 cu pasul 5, apoi numerele de la 100 la 200 cu pasul 10, întrucât primul apel este către contorul `fir1`, deci rezultatul afişat pe ecran ar trbui să fie:

```
0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100
100 110 120 130 140 150 160 170 180 190 200
```

In realitate însă, rezultatul obţinut va fi o intercalare de valori produse de cele două fire ce rulează simultan. La rulări diferite se pot obţine rezultate diferite deoarece timpul alocat fiecărui fir de execuţie poate să nu fie acelaşi, el fiind controlat de procesor într-o manieră "aparent" aleatoare. Un posibil rezultat al programului de mai sus:

```
0 100 5 110 10 120 15 130 20 140 25 150 160 170 180 190 200
30 35 40 45 50 55 60 65 70 75 80 85 90 95 100
```

12.2.2 Implementarea interfeţei Runnable

Ce facem însă când dorim să creăm o clasă care instanţiază fire de execuţie dar aceasta are deja o superclasă, ştiind că în Java nu este permisă moştenirea multiplă ?

```
  class FirExecutie extends Parinte, Thread // incorect !
```

In acest caz, nu mai putem extinde clasa `Thread` ci trebuie să implementăm direct interfaţa `Runnable`. Clasa `Thread` implementează ea însăşi interfaţa `Runnable` şi, din acest motiv, la extinderea ei obţineam implementarea indirectă a interfeţei. Aşadar, interfaţă `Runnable` permite unei clase să fie activă, fără a extinde clasa `Thread`.

Interfaţa `Runnable` se găseşte în pachetul `java.lang` şi este definită astfel:

```
public interface Runnable {
  public abstract void run();
}
```

Prin urmare, o clasă care instanţiază fire de execuţie prin implementarea interfeţei `Runnable` trebuie obligatoriu să implementeze metoda `run`. O astfel de clasă se mai numeşte *clasă activă* şi are următoarea structură:

```
public class ClasaActiva implements Runnable {

  public void run() {
    //Codul firului de executie
    ...
  }
}
```

Spre deosebire de modalitatea anterioară, se pierde însă tot suportul oferit de clasa `Thread`. Simpla instanţiere a unei clase care implemenează interfaţa `Runnable` nu creează nici un fir de execuţie, crearea acestora trebuind făcută explicit. Pentru a realiza acest lucru trebuie să instanţiem un obiect de tip `Thread` ce va reprezenta firul de execuţie propriu zis al cărui cod se gaseşte în clasa noastră. Acest lucru se realizează, ca pentru orice alt obiect, prin instrucţiunea `new`, urmată de un apel la un constructor al clasei `Thread`, însă nu la oricare dintre aceştia. Trebuie apelat constructorul care să primească drept argument o instanţă a clasei noastre. După creare, firul de execuţie poate fi lansat printr-un apel al metodei `start`.

```
  ClasaActiva obiectActiv = new ClasaActiva();
  Thread fir = new Thread(obiectActiv);
  fir.start();
```

Aceste operaţiuni pot fi făcute chiar în cadrul clasei noastre:

```
public class FirExecutie implements Runnable {

  private Thread fir = null;

  public FirExecutie()
    if (fir == null) {
      fir = new Thread(this);
```

```
      fir.start();
    }
  }

  public void run() {
    //Codul firului de executie
    ...
  }
}
```

Specificarea argumentului `this` în constructorul clasei `Thread` determină crearea unui fir de execuţie care, la lansarea sa, va apela metoda `run` din clasa curentă. Aşadar, acest constructor acceptă ca argument orice instanţă a unei clase "Runnable". Pentru clasa `FirExecutie` dată mai sus, lansarea firului va fi făcută automat la instanţierea unui obiect al clasei:

```
FirExecutie fir = new FirExecutie();
```

Atenţie

Metoda `run` nu trebuie apelată explicit, acest lucru realizându-se automat la apelul metodei `start`. Apelul explicit al metodei `run` nu va furniza nici o eroare, însă aceasta va fi executată ca orice altă metoda, şi nu separat într-un fir.

Să considerăm următorul exemplu în care creăm două fire de execuţie folosind interfaţa `Runnable`. Fiecare fir va desena figuri geometrice de un anumit tip, pe o suprafaţă de desenare de tip `Canvas`. Vom porni apoi două fire de execuţie care vor rula concurent, desenând figuri diferite, fiecare pe suprafaţa sa.

Listing 12.2: Folosirea interfeţei `Runnable`

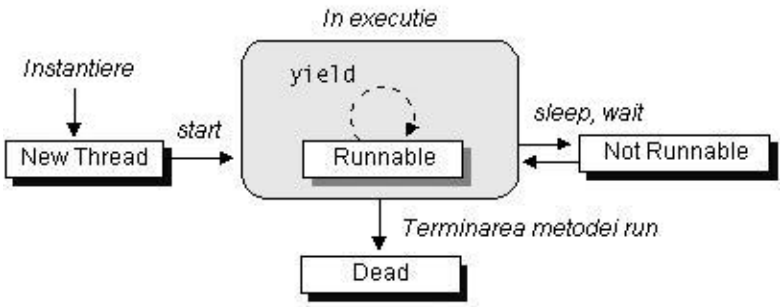
```
import java.awt.*;
import java.awt.event.*;

class Plansa extends Canvas implements Runnable {
  // Deoarece Plansa extinde Canvas,
```

```
// nu mai putem extinde clasa Thread

Dimension dim = new Dimension(300, 300);
Color culoare;
String figura;
int x=0, y=0, r=0;

public Plansa(String figura, Color culoare) {
  this.figura = figura;
  this.culoare = culoare;
}

public Dimension getPreferredSize() {
  return dim;
}

public void paint(Graphics g) {
  // Desenam un chenar
  g.setColor(Color.black);
  g.drawRect(0, 0, dim.width-1, dim.height-1);

  // Desenam figura la coordonatele calculate
  // de firul de executie
  g.setColor(culoare);
  if (figura.equals("patrat"))
    g.drawRect(x, y, r, r);
  else
  if (figura.equals("cerc"))
    g.drawOval(x, y, r, r);
}

public void update(Graphics g) {
  paint(g);
  // Supradefinim update ca sa nu mai
  // fie stearsa suprafata de desenare
}

public void run() {
  /* Codul firului de executie:
     Afisarea a 100 de figuri geometrice la pozitii
     si dimensiuni calculate aleator.
     Intre doua afisari, facem o pauza de 50 ms
  */

  for(int i=0; i<100; i++) {
```

```
      x = (int)(Math.random() * dim.width);
      y = (int)(Math.random() * dim.height);
      r = (int)(Math.random() * 100);
      try {
        Thread.sleep(50);
      } catch(InterruptedException e) {}
      repaint();
    }
  }
}

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

    // Cream doua obiecte active de tip Plansa
    Plansa p1 = new Plansa("patrat", Color.blue);
    Plansa p2 = new Plansa("cerc", Color.red);

    // Acestea extind Canvas, le plasam pe fereastra
    setLayout(new GridLayout(1, 2));
    add(p1);
    add(p2);
    pack();

    // Pornim doua fire de executie, care vor
    // actualiza desenul celor doua planse
    new Thread(p1).start();
    new Thread(p2).start();
  }
}

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

12.3 Ciclul de viaţă al unui fir de execuţie

Fiecare fir de execuţie are propriul său ciclu de viaţă: este creat, devine activ prin lansarea sa şi, la un moment dat, se termină. In continuare, vom analiza mai îndeaproape stările în care se poate găsi un fir de execuţie. Diagrama de mai jos ilustrează generic aceste stări precum şi metodele care provoaca tranziţia dintr-o stare în alta:



Aşadar, un fir de execuţie se poate găsi în una din următoarele patru stări:

- "New Thread"
- "Runnable"
- "Not Runnable"
- "Dead"

Starea "New Thread"
Un fir de execuţie se găseşte în această stare imediat după crearea sa, cu alte cuvinte după instanţierea unui obiect din clasa `Thread` sau dintr-o subclasă a sa.

```
Thread fir = new Thread(obiectActiv);
// fir se gaseste in starea "New Thread"
```

In această stare firul este "vid", el nu are alocate nici un fel de resurse sistem şi singura operaţiune pe care o putem executa asupra lui este lansarea în execuţie, prin metoda `start`. Apelul oricărei alte metode în afară de `start` nu are nici un sens şi va provoca o excepţie de tipul `IllegalThreadStateException`.

Starea "Runnable"

După apelul metodei `start` un fir va trece în starea "Runnable", adică va fi în execuţie.

```
fir.start();
//fir se gaseste in starea "Runnable"
```

Metoda `start` realizează următoarele operaţiuni necesare rulării firului de execuţie:

- Alocă resursele sistem necesare.
- Planifică firul de execuţie la procesor pentru a fi lansat.
- Apelează metoda `run` a obiectului activ al firului.

Un fir aflat în starea "Runnable" nu înseamnă neapărat că se găşeste efectiv în execuţie, adică instrucţiunile sale sunt interpretate de procesor. Acest lucru se întâmplă din cauza că majoritatea calculatoarelor au un singur procesor iar acesta nu poate rula simultan toate firele de execuţie care se gasesc în starea "Runnable". Pentru a rezolva aceasta problemă există o planificare care să partajeze dinamic şi corect procesorul între toate firele de execuţie care sunt în starea "Runnable". Aşadar, un fir care "rulează" poate să-şi aştepte de fapt rândul la procesor.

Starea "Not Runnable"

Un fir de execuţie poate ajunge în aceaată stare în una din următoarele situaţii:

- Este "adormit" prin apelul metodei `sleep`;
- A apelat metoda `wait`, aşteptând ca o anumită condiţie să fie satisfacută;

- Este blocat într-o operaţie de intrare/ieşire.

Metoda `sleep` este o metodă statică a clasei `Thread` care provoacă o pauză în timpul rulării firului curent aflat în execuţie, cu alte cuvinte îl "adoarme" pentru un timp specificat. Lungimea acestei pauze este specificată în milisecunde şi chiar nanosecunde. Intrucât poate provoca excepţii de tipul `InterruptedException`, apelul acestei metode se face într-un bloc de tip `try-cacth`:

```
try {
  // Facem pauza de o secunda
  Thread.sleep(1000);
} catch (InterruptedException e) {
  ...
}
```

Observaţi că metoda fiind statică apelul ei nu se face pentru o instanţă anume a clasei `Thread`. Acest lucru este foarte normal deoarece, la un moment dat, un singur fir este în execuţie şi doar pentru acesta are sens "adormirea" sa.

In intervalul în care un fir de execuţie "doarme", acesta nu va fi execut chiar dacă procesorul devine disponibil. După expirarea intervalului specificat firul revine în starea "Runnable" iar dacă procesorul este în continuare disponibil îsi va continua execuţia.

Pentru fiecare tip de intrare în starea "Not Runnable", există o secvenţă specifică de ieşire din starea repectivă, care readuce firul de execuţie în starea "Runnable". Acestea sunt:

- Dacă un fir de execuţie a fost "adormit", atunci el devine "Runnable" doar după scurgerea intervalului de timp specificat de instrucţiunea `sleep`.

- Dacă un fir de execuţie aşteaptă o anumită condiţie, atunci un alt obiect trebuie să îl informeze dacă acea condiţie este îndeplinită sau nu; acest lucru se realizează prin instrucţiunile `notify` sau `notifyAll` (vezi "Sincronizarea firelor de execuţie").

- Dacă un fir de execuţie este blocat într-o operaţiune de intrare/ieşire atunci el redevine "Runnable" atunci când acea operaţiune s-a terminat.

Starea "Dead"
Este starea în care ajunge un fir de execuţie la terminarea sa. Un fir nu poate fi oprit din program printr-o anumită metodă, ci trebuie să se termine în mod natural la încheierea metodei `run` pe care o execută. Spre deosebire de versiunile curente ale limbajului Java, în versiunile mai vechi exista metoda `stop` a clasei `Thread` care termina forţat un fir de execuţie, însă aceasta a fost eliminată din motive de securitate. Aşadar, un fir de execuţie trebuie să-şi "aranjeze" singur propria sa "moarte".

12.3.1 Terminarea unui fir de execuţie

După cum am vazut, un fir de execuţie nu poate fi terminat forţat de către program ci trebuie să-şi "aranjeze" singur terminarea sa. Acest lucru poate fi realizat în două modalităţi:

1. Prin scrierea unor metode `run` care să-şi termine execuţia în mod natural. La terminarea metodei `run` se va termina automat şi firul de execuţie, acesta intrând în starea Dead. Ambele exemple anteriorare se încadrează în această categorie.

   ```
   // Primul exemplu
   public void run() {
     for(int i = a; i <= b; i += pas)
       System.out.print(i + " " );
   }
   ```

   După afişarea numerelor din intervalul specificat, metoda se termină şi, odată cu ea, se va termina şi firul de execuţie repsectiv.

2. Prin folosirea unei variabile de terminare. In cazul când metoda `run` trebuie să execute o buclă infinită atunci aceasta trebuie controlată printr-o variabilă care să oprească ciclul atunci când dorim ca firul de execuţie să se termine. Uzual, vom folosi o variabilă membră a clasei care descrie firul de execuţie care fie este publică, fie este asociată cu o metodă publică care permite schimbarea valorii sale.

Să considerăm exemplul unui fir de execuţie care trebuie să numere secundele scurse până la apăsarea tastei Enter.

Listing 12.3: Folosirea unei variabile de terminare

```
import java.io.*;

class NumaraSecunde extends Thread {
  public int sec = 0;
  // Folosim o variabila de terminare
  public boolean executie = true;

  public void run() {
    while (executie) {
      try {
        Thread.sleep(1000);
        sec ++;
        System.out.print(".");
      } catch(InterruptedException e){}
    }
  }
}

public class TestTerminare {
  public static void main(String args[])
      throws IOException {

    NumaraSecunde fir = new NumaraSecunde();
    fir.start();

    System.out.println("Apasati tasta Enter");
    System.in.read();

    // Oprim firul de executie
    fir.executie = false;
    System.out.println("S-au scurs " + fir.sec + " secunde");
  }
}
```

Nu este necesară distrugerea explicită a unui fir de execuţie. Sistemul Java de colectare a "gunoiului" se ocupă de acest lucru. Setarea valorii `null` pentru variabila care referea instanţa firului de execuţie va uşura însă activitatea procesului *gc*.

Metoda `System.exit` va oprit forţat toate firele de execuţie şi va termina aplicaţia curentă.

Pentru a testa dacă un fir de execuţie a fost pornit dar nu s-a terminat încă putem folosi metoda **isAlive**. Metoda returnează:

- `true` - dacă firul este în una din stările "Runnable" sau "Not Runnable"
- `false` - dacă firul este în una din starile "New Thread" sau "Dead"

Intre stările "Runnable" sau "Not Runnable", repectiv "New Thread" sau "Dead" nu se poate face nici o diferenţiere.

```
NumaraSecunde fir = new NumaraSecunde();
// isAlive retuneaza false (starea este New Thread)

fir.start();
// isAlive retuneaza true (starea este Runnable)

fir.executie = false;
// isAlive retuneaza false (starea este Dead)
```

12.3.2 Fire de execuţie de tip "daemon"

Un proces este considerat în execuţie dacă conţine cel puţin un fir de execuţie activ. Cu alte cuvinte, la rularea unei aplicaţii, maşina virtuală Java nu se va opri decât atunci când nu mai există nici un fir de execuţie activ. De multe ori însă dorim să folosim fire care să realizeze diverse activităţi, eventual periodic, pe toată durata de execuţie a programului iar în momentul terminării acestuia să se termine automat şi firele respective. Aceste fire de execuţie se numesc *demoni*.

După crearea sa, un fir de execuţie poate fi făcut demon, sau scos din această stare, cu metoda **setDaemon**.

Listing 12.4: Crearea unui fir de excuţie de tip "daemon"

```
class Beeper implements Runnable {
  public void run() {
    while (true) {
      java.awt.Toolkit.getDefaultToolkit().beep();
      try {
        Thread.sleep(1000);
      } catch(InterruptedException e) {}
    }
```

```
  }
}

public class TestDaemon {
  public static void main(String args[])
      throws java.io.IOException {

    Thread t = new Thread(new Beeper());
    t.setDaemon(true);
    t.start();

    System.out.println("Apasati Enter...");
    System.in.read();

    // "Demonul" se termina automat
    // la terminarea aplicatiei
  }
}
```

12.3.3 Stabilirea priorităţilor de execuţie

Majoritatea calculatoarelor au un sigur procesor, ceea ce înseamnă că firele de execuţie trebuie să-şi împartă accesul la acel procesor. Execuţia într-o anumită ordine a mai multor fire de execuţie pe un număr limitat de procesoare se numeşte *planificare (scheduling)*. Sistemul Java de execuţie a programelor implementează un algoritm simplu, determinist de planificare, cunoscut sub numele de *planificare cu priorităţi fixate*.

Fiecare fir de execuţie Java primeşte la crearea sa o anumită prioritate. O prioritate este de fapt un număr întreg cu valori cuprinse între `MIN_PRIORITY` şi `MAX_PRIORITY`. Implicit, prioritatea unui fir nou creat are valoarea `NORM_PRIORITY`. Aceste trei constante sunt definite în clasa `Thread` astfel:

```
public static final int MAX_PRIORITY = 10;
public static final int MIN_PRIORITY = 1;
public static final int NORM_PRIORITY= 5;
```

Schimbarea ulterioară a priorităţii unui fir de execuţie se realizează cu metoda **setPriority** a clasei `Thread`.

La nivelul sistemului de operare, există două modele de lucru cu fire de execuţie:

- *Modelul cooperativ*, în care firele de execuţie decid când să cedeze procesorul; dezavantajul acestui model este că unele fire pot acapara procesorul, nepermiţând şi execuţia altora până la terminarea lor.
- *Modelul preemptiv*, în care firele de execuţie pot fi întrerupte oricând, după ce au fost lăsate să ruleze o perioadă, urmând să fie reluate după ce şi celelalte fire aflate în execuţie au avut acces la procesor; acest sistem se mai numeşte cu "cuante de timp", dezavantajul său fiind nevoia de a sincroniza accesul firelor la resursele comune.

Aşadar, îm modelul cooperativ firele de execuţie sunt responsabile cu partajarea timpului de execuţie, în timp ce în modelul preemptiv ele trebuie să partajeze resursele comune. Deoarece specificaţiile maşinii virtuale Java nu impun folosirea unui anumit model, programele Java trebuie scrise astfel încât să funcţioneze corect pe ambele modele. In continuare, vom mai detalia puţin aceste aspecte.

Planificatorul Java lucrează în modul urmator: dacă la un moment dat sunt mai multe fire de execuţie în starea "Runnable", adică sunt pregatite pentru a fi rulate, planificatorul îl va alege pe cel cu prioritatea cea mai mare pentru a-l executa. Doar când firul de execuţie cu prioritate maximă se termină, sau este suspendat din diverse motive, va fi ales un fir cu o prioritate mai mică. In cazul în care toate firele au aceeaşi prioritate ele sunt alese pe rând, după un algoritm simplu de tip "round-robin". De asemenea, dacă un fir cu prioritate mai mare decât firul care se execută la un moment dat solicită procesorul, atunci firul cu prioritate mai mare este imediat trecut în execuţie iar celalalt trecut în asteptare. Planificatorul Java nu va întrerupe însă un fir de execuţie în favoarea altuia de aceeasi prioritate, însă acest lucru îl poate face sistemul de operare în cazul în care acesta alocă procesorul în cuante de timp (un astfel de SO este Windows).

Aşadar, un fir de execuţie Java cedează procesorul în una din situatiile:

- un fir de execuţie cu o prioritate mai mare solicită procesorul;
- metoda sa `run` se termină;
- face explicit acest lucru apelând metoda `yield`;
- timpul alocat pentru execuţia să a expirat (pe SO cu cuante de timp).

Atenţie

In nici un caz corectitudinea unui program nu trebuie să se bazeze pe mecansimul de planificare a firelor de execuţie, deoarece acesta poate fi diferit de la un sistem de operare la altul.

Un fir de execuţie de lungă durată şi care nu cedează explicit procesorul la anumite intervale de timp astfel încât să poata fi executate şi celelalte fire de execuţie se numeşte fir de execuţie *egoist*. Evident, trebuie evitată scrierea lor întrucât acaparează pe termen nedefinit procesorul, blocând efectiv execuţia celorlalte fire de execuţie până la terminarea sa. Unele sistemele de operare combat acest tip de comportament prin metoda alocării procesorului în cuante de timp fiecărui fir de execuţie, însă nu trebuie să ne bazăm pe acest lucru la scrierea unui program. Un fir de execuţie trebuie să fie "corect" faţăde celelalte fire şi să cedeze periodic procesorul astfel încât toate să aibă posibilitatea de a se executa.

Listing 12.5: Exemplu de fir de execuţie "egoist"

```
class FirEgoist extends Thread {
  public FirEgoist(String name) {
    super(name);
  }
  public void run() {
    int i = 0;
    while (i < 100000) {
      // Bucla care acapareaza procesorul
      i ++;
      if (i % 100 == 0)
        System.out.println(getName() + " a ajuns la " + i);
      // yield();
    }
  }
}

public class TestFirEgoist {
  public static void main(String args[]) {
    FirEgoist s1, s2;
    s1 = new FirEgoist("Firul 1");
    s1.setPriority (Thread.MAX_PRIORITY);
    s2 = new FirEgoist("Firul 2");
    s2.setPriority (Thread.MAX_PRIORITY);
```

```
    s1.start();
    s2.start();
  }
}
```

Firul de execuţie $s1$ are prioritate maximă şi până nu-şi va termina execuţia nu-i va permite firului $s2$ să execute nici o instrucţiune, acaparând efectiv procesorul. Rezultatul va arăta astfel:

```
Firul 1 a ajuns la 100
Firul 1 a ajuns la 200
Firul 1 a ajuns la 300
...
Firul 1 a ajuns la 99900
Firul 1 a ajuns la 100000
Firul 2 a ajuns la 100
Firul 2 a ajuns la 200
...
Firul 2 a ajuns la 99900
Firul 2 a ajuns la 100000
```

Rezolvarea acestei probleme se face fie prin intermediul metodei statice `yield` a clasei `Thread`, care determină firul de execuţie curent să se oprească temporar, dând ocazia şi altor fire să se execute, fie prin "adormirea" temporară a firului curent cu ajutorul metodei `sleep`. Prin metoda `yield` un fir de execuţie nu cedează procesorul decât firelor de execuţie care au aceeaşi prioritate cu a sa şi nu celor cu priorităţi mai mici. Decomentând linia în care apelăm `yeld` din exemplul anterior, execuţia celor două fire se va intercala.

```
...
Firul 1 a ajuns la 31900
Firul 1 a ajuns la 32000
Firul 2 a ajuns la 100
Firul 1 a ajuns la 32100
Firul 2 a ajuns la 200
Firul 2 a ajuns la 300
...
```

12.3.4 Sincronizarea firelor de execuţie

Până acum am văzut cum putem crea fire de execuţie independente şi asincrone, cu alte cuvinte care nu depind în nici un fel de execuţia sau de rezultatele altor fire. Există însă numeroase situaţii când fire de execuţie separate, dar care rulează concurent, trebuie să comunice între ele pentru a accesa diferite resurse comune sau pentru a-şi transmite dinamic rezultatele "muncii" lor. Cel mai elocvent scenariu în care firele de execuţie trebuie să se comunice între ele este cunoscut sub numele de problema *producătorului/consumatorului*, în care producătorul generează un flux de date care este preluat şi prelucrat de către consumator.

Să considerăm de exemplu o aplicaţie Java în care un fir de execuţie (producătorul) scrie date într-un fişier în timp ce alt fir de execuţie (consumatorul) citeşte date din acelaşi fişier pentru a le prelucra. Sau, să presupunem că producătorul generează nişte numere şi le plasează, pe rând, într-un buffer iar consumatorul citeşte numerele din acel buffer pentru a le procesa. In ambele cazuri avem de-a face cu fire de execuţie concurente care folosesc o resursă comună: un fişier, respectiv o zonă de memorie şi, din acest motiv, ele trebuie sincronizate într-o manieră care să permită decurgerea normală a activităţii lor.

12.3.5 Scenariul producător / consumator

Pentru a înţelege mai bine modalitatea de sincronizare a două fire de execuţie să implementăm efectiv o problemă de tip producător/consumator. Să considerăm următoarea situaţie:

- *Producătorul* generează numerele întregi de la 1 la 10, fiecare la un interval neregulat cuprins între 0 şi 100 de milisecunde. Pe măsura ce le generează încearcă să le plaseze într-o zonă de memorie (o variabilă întreaga) de unde să fie citite de către consumator.

- *Consumatorul* va prelua, pe rând, numerele generate de către producător şi va afişa valoarea lor pe ecran.

Pentru a fi accesibilă ambelor fire de execuţie, vom încapsula variabila ce va contine numerele generate într-un obiect descris de clasa `Buffer` şi care va avea două metode **put** (pentru punerea unui numar în buffer) şi **get** (pentru obtinerea numarului din buffer).

Fără a folosi nici un mecanism de sincronizare clasa `Buffer` arată astfel:

Listing 12.6: Clasa `Buffer` fără sincronizare

```
class Buffer {
  private int number = -1;

  public int get() {
    return number;
  }

  public void put(int number) {
    this.number = number;
  }
}
```

Vom implementa acum clasele `Producator` şi `Consumator` care vor descrie cele două fire de execuţie. Ambele vor avea o referinţa comună la un obiect de tip `Buffer` prin intermediul căruia îşi comunică valorile.

Listing 12.7: Clasele `Producator` şi `Consumator`

```
class Producator extends Thread {
  private Buffer buffer;

  public Producator(Buffer b) {
    buffer = b;
  }

  public void run() {
    for (int i = 0; i < 10; i++) {
      buffer.put(i);
      System.out.println("Producatorul a pus:\t" + i);
      try {
        sleep((int)(Math.random() * 100));
      } catch (InterruptedException e) { }
    }
  }
}

class Consumator extends Thread {
  private Buffer buffer;

  public Consumator(Buffer b) {
    buffer = b;
```

```
  }

  public void run() {
    int value = 0;
    for (int i = 0; i < 10; i++) {
      value = buffer.get();
      System.out.println("Consumatorul a primit:\t" + value);
    }
  }
}

public class TestSincronizare1 {
  public static void main(String[] args) {
    Buffer b = new Buffer();
    Producator p1 = new Producator(b);
    Consumator c1 = new Consumator(b);
    p1.start();
    c1.start();
  }
}
```

După cum ne aşteptam, rezultatul rulării acestui program nu va rezolva nici pe departe problema propusă de noi, motivul fiind lipsa oricărei sincronizări între cele două fire de execuţie. Mai precis, rezultatul va fi ceva de forma:

```
Consumatorul a primit: -1
Consumatorul a primit: -1
Producatorul a pus:    0
Consumatorul a primit: 0
Consumatorul a primit: 0
Consumatorul a primit: 0
Consumatorul a primit: 0
Consumatorul a primit: 0
Consumatorul a primit: 0
Consumatorul a primit: 0
Consumatorul a primit: 0
Producatorul a pus:    1
Producatorul a pus:    2
Producatorul a pus:    3
Producatorul a pus:    4
Producatorul a pus:    5
```

```
Producatorul a pus:    6
Producatorul a pus:    7
Producatorul a pus:    8
Producatorul a pus:    9
```

Ambele fire de execuţie accesează resursa comună, adică obiectul de tip `Buffer`, într-o manieră haotică şi acest lucru se întâmpla din dou motive :

- Consumatorul nu aşteaptă înainte de a citi ca producătorul să genereze un număr şi va prelua de mai multe ori acelaşi număr.

- Producătorul nu aşteaptă consumatorul să preia numărul generat înainte de a produce un altul, în felul acesta consumatorul va "rata" cu siguranţă unele numere (în cazul nostru aproape pe toate).

Problema care se ridică în acest moment este: cine trebuie să se ocupe de sincronizarea celor două fire de execuţie : clasele `Producator` şi `Consumator` sau resursa comuna `Buffer` ? Răspunsul este evident: resursa comună `Buffer`, deoarece ea trebuie să permita sau nu accesul la conţinutul său şi nu firele de execuţie care o folosesc. In felul acesta efortul sincronizării este transferat de la producător/consumator la un nivel mai jos, cel al resursei critice.

Activităţile producătorului şi consumatorului trebuie sincronizate la nivelul resursei comune în două privinţe:

- Cele două fire de execuţie nu trebuie să acceseze simultan buffer-ul; acest lucru se realizează prin blocarea obiectului `Buffer` atunci când este accesat de un fir de execuţie, astfel încât nici nu alt fir de execuţie să nu-l mai poată accesa (vezi "Monitoare").

- Cele două fire de execuţie trebuie să se coordoneze, adică producătorul trebuie să găsească o modalitate de a "spune" consumatorului că a plasat o valoare în buffer, iar consumatorul trebuie să comunice producătorului că a preluat această valoare, pentru ca acesta să poată genera o alta. Pentru a realiza aceasta comunicare, clasa `Thread` pune la dispoziţie metodele `wait, notify, notifyAll`. (vezi "Semafoare").

Folosind sincronizarea clasa `Buffer` va arăta astfel:

Listing 12.8: Clasa `Buffer` cu sincronizare

```
class Buffer {
  private int number = -1;
  private boolean available = false;

  public synchronized int get() {
    while (!available) {
      try {
        wait();
        // Asteapta producatorul sa puna o valoare
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    available = false;
    notifyAll();
    return number;
  }

  public synchronized void put(int number) {
    while (available) {
      try {
        wait();
        // Asteapta consumatorul sa preia valoarea
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    this.number = number;
    available = true;
    notifyAll();
  }
}
```

Rezultatul obtinut va fi cel scontat:

```
Producatorul a pus:    0
Consumatorul a primit: 0
Producatorul a pus:    1
Consumatorul a primit: 1
...
Producatorul a pus:    9
Consumatorul a primit: 9
```

12.3.6 Monitoare

Definiţie

Un segment de cod ce gestionează o resursă comună mai multor de fire de execuţie separate concurente se numeşte *secţiune critică*. In Java, o secţiune critică poate fi un bloc de instrucţiuni sau o metodă.

Controlul accesului într-o secţiune critică se face prin cuvântul cheie **synchronized**. Platforma Java asociază un monitor ("lacăt") fiecărui obiect al unui program aflat în execuţie. Acest monitor va indica dacă resursa critică este accesată de vreun fir de execuţie sau este liberă, cu alte cuvinte "monitorizează" resursa respectivă. In cazul în care este accesată, va pune un lacăt pe aceasta, astfel încât să împiedice accesul altor fire de execuţie la ea. In momentul când resursa este eliberată "lacătul" va fi eliminat, pentru a permite accesul altor fire de execuţie.

In exemplul de tip producător/consumator de mai sus, secţiunile critice sunt metodele `put` şi `get` iar resursa critică comună este obiectul `buffer`. Consumatorul nu trebuie să acceseze buffer-ul când producatorul tocmai pune o valoare în el, iar producătorul nu trebuie să modifice valoarea din buffer în momentul când aceasta este citită de către consumator.

```
public synchronized int get() {
  ...
}
public synchronized void put(int number) {
  ...
}
```

Să observam că ambele metode au fost declarate cu modificatorul `synchronized`. Cu toate acestea, sistemul asociază un monitor unei instanţe a clasei `Buffer` şi nu unei metode anume. In momentul în care este apelată o metodă sincronizată, firul de execuţie care a facut apelul va bloca obiectul a cărei metodă o accesează, ceea ce înseamnă că celelalte fire de execuţie nu vor mai putea accesa resursele critice ale acelui obiect. Acesta este un lucru logic, deoarece mai multe secţiuni critice ale unui obiect gestionează de fapt o singură resursă critică.

In exemplul nostru, atunci când producatorul apelează metoda `put` pentru a scrie un număr, va bloca tot obiectul `buffer`, astfel că firul de execuţie

consumator nu va avea acces la metoda `get`, şi reciproc.

```
public synchronized void put(int number) {
  // buffer blocat de producator
  ...
  // buffer deblocat de producator
}
public synchronized int get() {
  // buffer blocat de consumator
  ...
  // buffer deblocat de consumator
}
```

Monitoare fine

Adeseori, folosirea unui monitor pentru întreg obiectul poate fi prea restrictivă. De ce să blocăm toate resursele unui obiect dacă un fir de execuţie nu doreşte decât accesarea uneia sau a câtorva dintre ele ? Deoarece orice obiect are un monitor, putem folosi obiecte fictive ca lacăte pentru fiecare din resursele obiectului nostru, ca în exemplul de mai jos:

```
class MonitoareFine {
  //Cele doua resurse ale obiectului
  Resursa x, y;

  //Folosim monitoarele a doua obiecte fictive
  Object xLacat = new Object(),
         yLacat = new Object();

  public void metoda() {
    synchronized(xLacat) {
      // Accesam resursa x
    }
    // Cod care nu foloseste resursele comune
    ...
    synchronized(yLacat) {
      // Accesam resursa y
    }
```

```
    ...
    synchronized(xLacat) {
      synchronized(yLacat) {
        // Accesam x si y
      }
    }
    ...
    synchronized(this) {
      // Accesam x si y
    }
  }
}
```

Metoda de mai sus nu a fost declarată cu `synchronized` ceea ce ar fi determinat blocarea tuturor resurselor comune la accesarea obiectului respectiv de un fir de execuţie, ci au fost folosite monitoarele unor obiecte fictive pentru a controla folosirea fiecărei resursă în parte.

12.3.7 Semafoare

Obiectul de tip `Buffer` din exemplul anterior are o variabilă membră privată numită *number*, în care este memorat numărul pe care îl comunică producatorul şi pe care îl preia consumatorul. De asemenea, mai are o variabilă privată logică *available* care ne dă starea buffer-ului: dacă are valoarea `true` înseamnă că producătorul a pus o valoare în buffer şi consumatorul nu a preluat-o înca; dacă este `false`, consumatorul a preluat valoarea din buffer dar producătorul nu a pus deocamdată alta la loc. Deci, la prima vedere, metodele clasei `Buffer` ar trebui să arate astfel:

```
  public synchronized int get() {
    while (!available) {
      // Nimic - asteptam ca variabila sa devina true
    }
    available = false;
    return number;
  }
  public synchronized int put(int number) {
    while (available) {
```

```
      // Nimic - asteptam ca variabila sa devina false
    }
    available = true;
    this.number = number;
  }
```

Varianta de mai sus, deşi pare corectă, nu este. Aceasta deoarece implementarea metodelor este "selfish", cele două metode îsi asteaptă în mod egoist condiţia de terminare. Ca urmare, corectitudinea funcţionării va depinde de sistemul de operare pe care programul este rulat, ceea ce reprezintă o greşeală de programare.

Punerea corectă a unui fir de execuţie în asteptare se realizează cu metoda **wait** a clasei `Thread`, care are următoarele forme:

```
void wait( )
void wait( long timeout )
void wait( long timeout, long nanos )
```

După apelul metodei `wait`, firul de execuţie curent eliberează monitorul asociat obiectului respectiv şi aşteaptă ca una din următoarele condiţii să fie îndeplinită:

- Un alt fir de execuţie informează pe cei care "aşteaptă" la un anumit monitor să se "trezească" - acest lucru se realizează printr-un apel al metodei **notifyAll** sau **notify**.
- Perioada de aştepatare specificată a expirat.

Metoda `wait` poate produce excepţii de tipul `InterruptedException`, atunci când firul de execuţie care aşteaptă (este deci în starea "Not Runnable") este întrerupt din aşteptare şi trecut forţat în starea "Runnable", deşi condiţia aşteptată nu era încă îndeplinită.

Metoda `notifyAll` informează toate firele de execuţie care sunt în asteptare la monitorul obiectului curent îndeplinirea condiţiei pe care o aşteptau. Metoda `notify` informează doar un singur fir de execuţie, specificat ca argument.

Reamintim varianta corectă a clasei `Buffer`:

Listing 12.9: Folosirea semafoarelor

```
class Buffer {
  private int number = -1;
```

```
  private boolean available = false;

  public synchronized int get() {
    while (!available) {
      try {
        wait();
        // Asteapta producatorul sa puna o valoare
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    available = false;
    notifyAll();
    return number;
  }

  public synchronized void put(int number) {
    while (available) {
      try {
        wait();
        // Asteapta consumatorul sa preia valoarea
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    this.number = number;
    available = true;
    notifyAll();
  }
}
```

12.3.8 Probleme legate de sincronizare

Din păcate, folosirea monitoarelor ridică şi unele probleme. Să analizăm câteva dintre ele şi posibilele lor soluţii:

Deadlock

Deadlock-ul este o problemă clasică într-un mediu în care rulează mai multe fire de execuţie şi constă în faptul că, la un moment dat, întreg procesul se poate bloca deoarece unele fire aşteaptă deblocarea unor monitoare care nu se vor debloca niciodată. Există numeroase exemple în acest sens, cea mai cunoscută fiind "Problema filozofilor". Reformulată, să ne imaginăm două persoane "A" şi "B" (fire de execuţie) care stau la aceeaşi masă şi tre-

buie să folosească în comun cuţitul şi furculiţa (resursele comune) pentru a mânca. Evident, cele două persoane doresc obţinerea ambelor resurse. Să presupunem că "A" a oţinut cuţitul şi "B" furculiţa. Firul "A" se va bloca în aşteptarea eliberării furculiţei iar firul "A" se va bloca în aştepatrea eliberării cuţitului, ceea ce conduce la starea de "deadlock". Deşi acest exemplu este desprins de realitate, există numeroase situaţii în care fenomenul de "deadlock" se poate manifesta, multe dintre acestea fiind dificil de detectat.

Există câteva reguli ce pot fi aplicate pentru evitarea deadlock-ului:

- Firele de execuţie să solicite resursele în aceeaşi ordine. Această abordare elimină situaţiile de aşteptare circulară.
- Folosirea unor monitoare care să controleze accesul la un grup de resurse. In cazul nostru, putem folosi un monitor "tacâmuri" care trebuie blocat înainte de a cere furculiţa sau cuţitul.
- Folosirea unor variabile care să informeze disponibilitatea resurselor fără a bloca monitoarele asociate acestora.
- Cel mai importat, conceperea unei arhitecturi a sistemului care să evite pe cât posibil apariţia unor potenţiale situaţii de deaslock.

Variabile volatile

Cuvântul cheie `volatile` a fost introdus pentru a controla unele aspecte legate de optimizările efectuate de unele compilatoare. Să considerăm următorul exemplu:

```
class TestVolatile {
  boolean test;
  public void metoda() {
    test = false;
    // *
    if (test) {
    // Aici se poate ajunge...
  }
}
}
```

Un compilator care optimizează codul, poate decide că variabila *test* fiind setată pe `false`, corpul *if*-ului nu se va executa şi să excludă secvenţa respectivă din rezultatul compilării. Dacă această clasă ar fi însă accesată de mai multe fire de execuţie, variabile *test* ar putea fi setată pe `true` de un alt fir, exact între instrucţiunile de atribuire şi `if` ale firului curent.

Declararea unei variabile cu modificatorul `volatile` informează compilatorul să nu optimizeze codul în care aceasta apare, previzionând valoarea pe care variabila o are la un moment dat.

Fire de execuţie inaccesibile

Uneori firele de execuţie sunt blocate din alte motive decât aşteptarea la un monitor, cea mai frecventă situaţie de acest tip fiind operaţiunile de intrare/ieşire (IO) blocante. Când acest lucru se întâmplă celelalte fire de execuţie trebuie să poată accesa în continuare obiectul. Dar dacă operaţiunea IO a fost făcută într-o metodă sincronizată, acest lucru nu mai este posibil, monitorul obiectului fiind blocat de firul care aşteaptă de fapt să realizeze operaţia de intrare/ieşire. Din acest motiv, operaţiile IO nu trebuie făcute în metode sincronizate.

12.4 Gruparea firelor de execuţie

Gruparea firelor de execuţie pune la dispoziţie un mecanism pentru manipularea acestora ca un tot şi nu individual. De exemplu, putem să pornim sau să suspendăm toate firele dintr-un grup cu un singur apel de metodă. Gruparea firelor de execuţie se realizează prin intermediul clasei **ThreadGroup**.

Fiecare fir de execuţie Java este membru al unui grup, indiferent dacă specificăm explicit sau nu acest lucru. Afilierea unui fir la un anumit grup se realizează la crearea sa şi devine permanentă, în sensul că nu vom putea muta un fir dintr-un grup în altul, după ce acesta a fost creat. In cazul în care creăm un fir folosind un constructor care nu specifică din ce grup face parte, el va fi plasat automat în acelaşi grup cu firul de execuţie care l-a creat. La pornirea unui program Java se creează automat un obiect de tip `ThreadGroup` cu numele **main**, care va reprezenta grupul tuturor firelor de execuţie create direct din program şi care nu au fost ataşate explicit altui grup. Cu alte cuvinte, putem să ignorăm complet plasarea firelor de execuţie în grupuri şi să lăsăm sistemul să se ocupe cu aceasta, adunându-le pe toate

în grupul `main`.

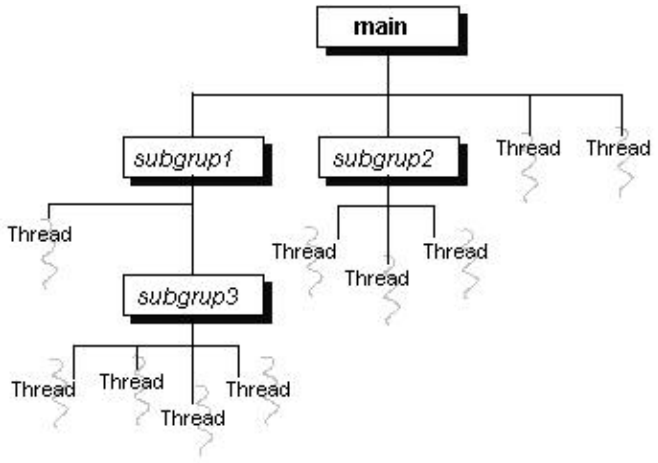
Există situaţii însă când gruparea firelor de execuţie poate uşura substanţial manevrarea lor. Crearea unui fir de execuţie şi plasarea lui într-un grup (altul decât cel implicit) se realizează prin următorii constructori ai clasei `Thread`:

```
public Thread(ThreadGroup group, Runnable target)
public Thread(ThreadGroup group, String name)
public Thread(ThreadGroup group, Runnable target, String name)
```

Fiecare din aceşti costructori creează un fir de execuţie, îl iniţializează şi îl plasează într-un grup specificat ca argument. Pentru a afla cărui grup aparţine un anumit fir de execuţie putem folosi metoda `getThreadGroup` a clasei `Thread`. In exemplul următor vor fi create două grupuri, primul cu două fire de execuţie iar al doilea cu trei:

```
ThreadGroup grup1 = new ThreadGroup("Producatori");
Thread p1 = new Thread(grup1, "Producator 1");
Thread p2 = new Thread(grup1, "Producator 2");

ThreadGroup grup2 = new ThreadGroup("Consumatori");
Thread c1 = new Thread(grup2, "Consumator 1");
Thread c2 = new Thread(grup2, "Consumator 2");
Thread c3 = new Thread(grup2, "Consumator 3");
```

Un grup poate avea ca părinte un alt grup, ceea ce înseamnă că firele de execuţie pot fi plasate într-o ierarhie de grupuri, în care rădăcina este grupul implicit `main`, ca în figura de mai jos:



Să considerăm un exemplu îm care listăm firele de execuţie active:

Listing 12.10: Folosirea clasei `ThreadGroup`

```
public class TestThreadGroup {

  static class Dummy implements Runnable {
    public void run() {
      while (true)
        Thread.yield();
    }
  }

  public static void main(String args[]) {

    // Cream o fereastra pentru a fi create
    // automat firele de executie din AWT
    java.awt.Frame f = new java.awt.Frame("Test");

    // Cream un fir propriu
    new Thread(new Dummy(), "Fir de test").start();

    // Obtinem o referinta la grupul curent
    Thread firCurent = Thread.currentThread();
    ThreadGroup grupCurent = firCurent.getThreadGroup();

    // Aflam numarul firelor de executie active
    int n = grupCurent.activeCount();

    // Enumeram firele din grup
    Thread[] lista = new Thread[n];
    grupCurent.enumerate(lista);

    // Le afisam
    for (int i=0; i < n; i++)
      System.out.println("Thread #" + i + "=" +
        lista[i].getName());
  }
}
```

12.5 Comunicarea prin fluxuri de tip "pipe"

O modalitate deosebit de utilă prin care două fire de execuţie pot comunica este realizată prin intermediul *canalelor de comunicatii (pipes)*. Acestea sunt implementate prin fluxuri descrise de clasele:

- **PipedReader, PipedWriter** - pentru caractere, respectiv
- **PipedOutputStream, PipedInputStream** - pentru octeţi.

Fluxurile "pipe" de ieşire şi cele de intrare pot fi conectate pentru a efectua transmiterea datelor. Acest lucru se realizează uzual prin intemediul constructorilor:

```
public PipedReader(PipedWriterpw)
public PipedWriter(PipedReaderpr)
```

In cazul în care este folosit un constructor fără argumente, conectarea unui flux de intrare cu un flux de ieşire se face prin metoda **connect**:

```
public void connect(PipedWriterpw)
public void connect(PipedReaderpr)
```

Intrucât fluxurile care sunt conectate printr-un pipe trebuie să execute simultan operaţii de scriere/citire, folosirea lor se va face din cadrul unor fire de execuţie.

Funcţionarea obicetelor care instanţiază `PipedWriter` şi `PipedReader` este asemănătoare cu a canalelor de comunicare UNIX (pipes). Fiecare capăt al unui canal este utilizat dintr-un fir de execuţie separat. La un capăt se scriu caractere, la celălalt se citesc. La citire, dacă nu sunt date disponibile firul de execuţie se va bloca până ce acestea vor deveni disponibile. Se observă că acesta este un comportament tipic producător-consumator asincron, firele de execuţie comunicând printr-un canal.

Realizarea conexiunii se face astfel:

```
PipedWriter pw1 = new PipedWriter();
PipedReader pr1 = new PipedReader(pw1);
// sau
PipedReader pr2 = new PipedReader();
PipedWriter pw2 = new PipedWriter(pr2);
// sau
```

```
PipedReader pr = new PipedReader();
PipedWriter pw = new PipedWirter();
pr.connect(pw) //echivalent cu
pw.connect(pr);
```

Scrierea şi citirea pe/de pe canale se realizează prin metodele uzuale `read` şi `write`, în toate formele lor.

Să reconsiderăm acum exemplul producător/consumator prezentat anterior, folosind canale de comunicaţie. Producătorul trimite datele printr-un flux de ieşire de tip `DataOutputStream` către consumator, care le primeşte printr-un flux de intrare de tip `DataInputStream`. Aceste două fluxuri vor fi interconectate prin intermediul unor fluxuri de tip "pipe".

Listing 12.11: Folosirea fluxurilor de tip "pipe"

```
import java.io.*;

class Producator extends Thread {
  private DataOutputStream out;

  public Producator(DataOutputStream out) {
    this.out = out;
  }

  public void run() {
    for (int i = 0; i < 10; i++) {
      try {
        out.writeInt(i);
      } catch (IOException e) {
        e.printStackTrace();
      }
      System.out.println("Producatorul a pus:\t" + i);
      try {
        sleep((int)(Math.random() * 100));
      } catch (InterruptedException e) { }
    }
  }
}

class Consumator extends Thread {
  private DataInputStream in;

  public Consumator(DataInputStream in) {
    this.in = in;
```

```
  }

  public void run() {
    int value = 0;
    for (int i = 0; i < 10; i++) {
      try {
        value = in.readInt();
      } catch (IOException e) {
        e.printStackTrace();
      }
      System.out.println("Consumatorul a primit:\t" + value);
    }
  }
}


public class TestPipes {
  public static void main(String[] args)
      throws IOException {

    PipedOutputStream pipeOut = new PipedOutputStream();
    PipedInputStream pipeIn = new PipedInputStream(pipeOut);

    DataOutputStream out = new DataOutputStream(pipeOut);
    DataInputStream in = new DataInputStream(pipeIn);

    Producator p1 = new Producator(out);
    Consumator c1 = new Consumator(in);

    p1.start();
    c1.start();
  }
}
```

12.6 Clasele Timer şi TimerTask

Clasa `Timer` oferă o facilitate de a planifica diverse acţiuni pentru a fi realizate la un anumit moment de către un fir de execuţie ce rulează în fundal. Acţiunile unui obiect de tip `Timer` sunt implementate ca instanţe ale clasei `TimerTask` şi pot fi programate pentru o singură execuţie sau pentru execuţii repetate la intervale regulate.

Paşii care trebuie făcuţi pentru folosirea unui timer sunt:

- Crearea unei subclase `Actiune` a lui `TimerTask` şi supreadefinirea metodei `run` ce va conţine acţiunea pe care vrem să o planificăm. După cum vom vedea, pot fi folosite şi clase anonime.

- Crearea unui fir de execuţie prin instanţierea clasei `Timer`;

- Crearea unui obiect de tip `Actiune`;

- Planificarea la execuţie a obiectuluii de tip `Actiune`, folosind metoda `schedule` din clasa `Timer`;

Metodele de planificare pe care le avem la dispoziţie au următoarele formate:

```
schedule(TimerTask task, Date time)
schedule(TimerTask task, long delay, long period)
schedule(TimerTask task, Date time, long period)
scheduleAtFixedRate(TimerTask task, long delay, long period)
scheduleAtFixedRate(TimerTask task, Date time, long period)
```

unde, *task* descrie acţiunea ce se va executa, *delay* reprezintă întârzierea faţa de momentul curent după care va începe execuţia, *time* momentul exact la care va începe execuţia iar *period* intervalul de timp între două execuţii.

După cum se observă, metodele de planificare se împart în două categorii:

- **schedule** - planificare cu întârziere fixă: dacă dintr-un anumit motiv acţiunea este întârziată, următoarele acţiuni vor fi şi ele întârziate în consecinţă;

- **scheduleAtFixedRate** - planificare cu număr fix de rate: dacă dintr-un anumit motiv acţiunea este întârziată, următoarele acţiuni vor fi executată mai repede, astfel încât numărul total de acţiuni dintr-o perioadă de timp să fie tot timpul acelaşi;

Un timer se va opri natural la terminarea metodei sale `run` sau poate fi oprit forţat folosind metoda `cancel`. După oprirea sa el nu va mai putea fi folosit pentru planificarea altor acţiuni. De asemenea, metoda `System.exit` va oprit forţat toate firele de execuţie şi va termina aplicaţia curentă.

Listing 12.12: Folosirea claselor `Timer` şi `TimerTask`

```
import java.util.*;
import java.awt.*;

class Atentie extends TimerTask {
  public void run() {
    Toolkit.getDefaultToolkit().beep();
    System.out.print(".");
  }
}

class Alarma extends TimerTask {
  public String mesaj;
  public Alarma(String mesaj) {
    this.mesaj = mesaj;
  }
  public void run() {
    System.out.println(mesaj);
  }
}

public class TestTimer {
  public static void main(String args[]) {

    // Setam o actiune repetitiva, cu rata fixa
    final Timer t1 = new Timer();
    t1.scheduleAtFixedRate(new Atentie(), 0, 1*1000);

    // Folosim o clasa anonima pentru o alta actiune
    Timer t2 = new Timer();
    t2.schedule(new TimerTask() {
      public void run() {
        System.out.println("S-au scurs 10 secunde.");
        // Oprim primul timer
        t1.cancel();
      }
    }, 10*1000);

    // Setam o actiune pentru ora 22:30
    Calendar calendar = Calendar.getInstance();
    calendar.set(Calendar.HOUR_OF_DAY, 22);
    calendar.set(Calendar.MINUTE, 30);
    calendar.set(Calendar.SECOND, 0);
    Date ora = calendar.getTime();
```

```
    Timer t3 = new Timer();
    t3.schedule(new Alarma("Toti copiii la culcare!"), ora);
  }
}
```

Capitolul 13

Programare în reţea

13.1 Introducere

Programarea în reţea implică trimiterea de mesaje şi date între aplicaţii ce rulează pe calculatoare aflate într-o reţea locală sau conectate la Internet. Pachetul care oferă suport pentru scrierea aplicatiilor de reţea este **java.net**. Clasele din acest pachet oferă o modalitate facilă de programare în reţea, fără a fi nevoie de cunoştinţe prealabile referitoare la comunicarea efectivă între calculatoare. Cu toate acestea, sunt necesare câteva noţiuni fundamentale referitoare la reţele cum ar fi: protocol, adresa IP, port, socket.

Ce este un protocol ?
Un *protocol* reprezintă o convenţie de reprezentare a datelor folosită în comunicarea între două calculatoare. Având în vedere faptul că orice informaţie care trebuie trimisă prin reţea trebuie serializată astfel încât să poată fi transmisă secvenţial, octet cu octet, către destinaţie, era nevoie de stabilirea unor convenţii (protocoale) care să fie folosite atât de calculatorul care trimite datele cât şi de cel care le primeşte, pentru a se "înţelege" între ele.

Două dintre cele mai utilizate protocoale sunt **TCP** şi **UDP**.

- *TCP (Transport Control Protocol)* este un protocol ce furnizează un flux sigur de date între două calculatoare aflate în reţea. Acest protocol asigură stabilirea unei conexiuni permanente între cele două calculatoare pe parcursul comunicaţiei.
- *UDP (User Datagram Protocol)* este un protocol bazat pe pachete inde-

pendente de date, numite *datagrame*, trimise de la un calculator către altul fără a se garanta în vreun fel ajungerea acestora la destinaţie sau ordinea în care acestea ajung. Acest protocol nu stabileşte o conexiună permantă între cele două calculatoare.

Cum este identificat un calculator în reţea ?

Orice calculator conectat la Internet este identificat în mod unic de adresa sa **IP** (IP este acronimul de la *Internet Protocol*). Aceasta reprezintă un număr reprezentat pe 32 de biţi, uzual sub forma a 4 octeţi, cum ar fi de exemplu: `193.231.30.131` şi este numit adresa IP *numerică*. Corespunzătoare unei adrese numerice exista şi o adresa IP *simbolică*, cum ar fi `thor.infoiasi.ro` pentru adresa numerică anterioară.

De asemenea, fiecare calculator aflat într-o reţea locală are un nume unic ce poat fi folosit la identificarea locală a acestuia.

Clasa Java care reprezintă noţiunea de adresă IP este **InetAddress**.

Ce este un port ?

Un calculator are în general o singură legătură fizică la reţea. Orice informaţie destinată unei anumite maşini trebuie deci să specifice obligatoriu adresa IP a acelei maşini. Insă pe un calculator pot exista concurent mai multe procese care au stabilite conexiuni în reţea, asteptând diverse informaţii. Prin urmare, datele trimise către o destinaţie trebuie să specifice pe lânga adresa IP a calculatorului şi procesul către care se îndreaptă informaţiile respective. Identificarea proceselor se realizează prin intermdiul **porturilor**.

Un *port* este un număr pe 16 biţi care identifică în mod unic procesele care rulează pe o anumită masină. Orice aplicaţie care realizează o conexiune în reţea va trebui să ataşeze un număr de port acelei conexiuni. Valorile pe care le poate lua un număr de port sunt cuprinse între 0 şi 65535 (deoarece sunt numere reprezentate pe 16 biţi), numerele cuprinse între 0 şi 1023 fiind însă rezervate unor servicii sistem şi, din acest motiv, nu trebuie folosite în aplicaţii.

Clase de bază din java.net

Clasele din `java.net` permit comunicarea între procese folosind protocoalele

TCP şi UDP şi sunt prezentate în tabelul de mai jos.

TCP	UDP
URL	DatagramPacket
URLConnection	DatagramSocket
Socket	MulticastSocket
ServerSocket	

13.2 Lucrul cu URL-uri

Termenul *URL* este acronimul pentru *Uniform Resource Locator* şi reprezintă o referinţă (adresă) la o resursă aflată pe Internet. Aceasta este în general un fişier reprezentând o pagină Web, un text, imagine, etc., însă un URL poate referi şi interogări la baze de date, rezultate ale unor comenzi executate la distanţă, etc. Mai jost, sunt prezentate câteva exemple de URL-uri sunt:

```
http://java.sun.com
http://students.infoiasi.ro/index.html
http://www.infoiasi.ro/~acf/imgs/taz.gif
http://www.infoiasi.ro/~acf/java/curs/9/prog_retea.html#url
```

După cum se observă din exemplele de mai sus, un URL are două componente principale:

- Identificatorul protocolului folosit (http, ftp, etc);
- Numele resursei referite. Acesta are următoarele componente:
  - Numele calculatorului gazdă (`www.infoiasi.ro`).
  - Calea completă spre resursa referită (`acf/java/curs/9/prog_retea.html`). Notaţia `user` semnifică uzual subdirectorul `html` al directorului rezervat pe un server Web utilizatorului specificat (`HOME`). In cazul în care este specificat doar un director, fişierul ce reprezintă resursa va fi considerat implicit `index.html`.
  - Opţional, o referinţă de tip *anchor* în cadrul fişierului referit (`#url`).
  - Opţional, portul la care să se realizeze conexiunea.

Clasa care permite lucrul cu URL-uri este **java.net.URL**. Aceasta are mai mulţi constructori pentru crearea de obiecte ce reprezintă referinţe către resurse aflate în reţea, cel mai uzual fiind cel care primeşte ca parametru un şir de caractere. In cazul în care şirul nu reprezintă un URL valid va fi aruncată o excepţie de tipul `MalformedURLException`.

```
try {
  URL adresa = new URL("http://xyz.abc");
} catch (MalformedURLException e) {
  System.err.println("URL invalid !\n"  + e);
}
```

Un obiect de tip URL poate fi folosit pentru:

- Aflarea informaţiilor despre resursa referită (numele calculatorului gazdă, numele fişierului, protocolul folosit. etc).
- Citirea printr-un flux a continutului fişierului respectiv.
- Conectarea la acel URL pentru citirea şi scrierea de informaţii.

Citirea conţinutului unui URL

Orice obiect de tip `URL` poate returna un flux de intrare de tip `InputStream` pentru citirea continutului său. Secvenţa standard pentru această operatiune este prezentată în exemplul de mai jos, în care afişam conţinutul resursei specificată la linia de comandă. Dacă nu se specifică mici un argument, va fi afişat fişierul `index.html` de la adresa: `http://www.infoiasi.ro`.

Listing 13.1: Citirea conţinutului unui URL

```
import java.net.*;
import java.io.*;

public class CitireURL {
  public static void main(String[] args)
      throws IOException{
    String adresa = "http://www.infoiasi.ro";
    if (args.length > 0)
      adresa = args[0];
```

```
    BufferedReader br = null;
    try {
      URL url = new URL(adresa);
      InputStream in = url.openStream();
      br = new BufferedReader(new InputStreamReader(in));
      String linie;
      while ((linie = br.readLine()) != null) {
        // Afisam linia citita
        System.out.println(linie);
      }
    } catch(MalformedURLException e) {
      System.err.println("URL invalid !\n" + e);
    } finally {
      br.close();
    }
  }
}
```

Conectarea la un URL

Se realizează prin metoda **openConnection** ce stabileşte o conexiune bidirecţională cu resursa specificată. Această conexiune este reprezentată de un obiect de tip **URLConnection**, ce permite crearea atât a unui flux de intrare pentru citirea informaţiilor de la URL-ul specificat, cât şi a unui flux de ieşire pentru scrierea de date către acel URL. Operaţiunea de trimitere de date dintr-un program către un URL este similară cu trimiterea de date dintr-un formular de tip `FORM` aflat într-o pagină HTML. Metoda folosită pentru trimitere este `POST`.

In cazul trimiterii de date, obiectul URL este uzual un proces ce rulează pe serverul Web referit prin URL-ul respectiv (jsp, servlet, cgi-bin, php, etc).

13.3 Socket-uri

Definiţie

Un *socket (soclu)* este o abstracţiune software folosită pentru a reprezenta fiecare din cele două "capete" ale unei conexiuni între două procese ce rulează într-o reţea. Fiecare socket este ataşat unui port astfel încât să poată identifica unic programul căruia îi sunt destinate datele.

Socket-urile sunt de două tipuri:

- TCP, implementate de clasele `Socket` şi `ServerSocket`;
- UDP, implementate de clasa `DatagramSocket`.

O aplicaţie de reţea ce foloseşte socket-uri se încadrează în modelul **client/server** de concepere a unei aplicaţii. In acest model aplicaţia este formată din două categorii distincte de programe numite *servere*, respectiv *clienţi*.

Programele de tip server sunt cele care oferă diverse servicii eventualilor clienţi, fiind în stare de aşteptare atâta vreme cât nici un client nu le solicită serviciile. Programele de tip client sunt cele care iniţiază conversaţia cu un server, solicitând un anumit serviciu. Uzual, un server trebuie să fie capabil să trateze mai mulţi clienti simultan şi, din acest motiv, fiecare cerere adresată serverului va fi tratată într-un fir de execuţie separat.

Incepând cu versiunea 1.4 a platformei standard Java, există o clasă utilitară care implementează o pereche de tipul *(adresa IP, număr port)*. Aceasta este **InetSocketAddress** (derivată din `SocketAddress`), obiectele sale fiind utilizate de constructori şi metode definite în cadrul claselor ce descriu socketuri, pentru a specifica cei doi parametri necesari identificării unui proces care trimite sau recepţionează date în reţea.

13.4 Comunicarea prin conexiuni

In acest model se stabileşte o conexiune TCP între o aplicaţie client şi o aplicaţie server care furnizează un anumit serviciu. Avantajul protocolul TCP/IP este că asigură realizarea unei comunicări stabile, permanente în reţea, existând siguranţa că informaţiile trimise de un proces vor fi recepţionate corect şi complet la destinaţie sau va fi semnalată o excepţie în caz contrar.

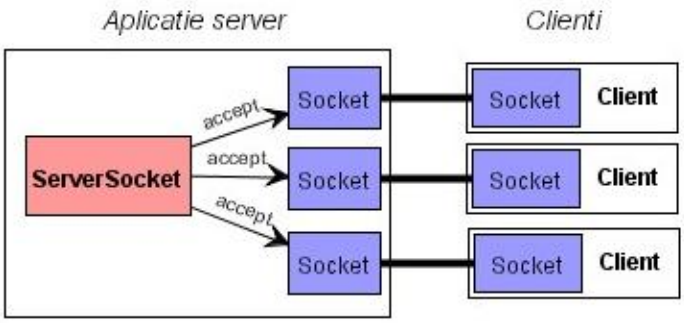
Legătura între un client şi un server se realizează prin intermediul a două obiecte de tip **Socket**, câte unul pentru fiecare capăt al "canalului" de comunicaţie dintre cei doi. La nivelul clientului crearea socketului se realizează specificând adresa IP a serverului şi portul la care rulează acesta, constructorul uzual folosit fiind:

```
Socket(InetAddress address, int port)
```

La nivelul serverului, acesta trebuie să creeze întâi un obiect de tip **ServerSocket**. Acest tip de socket nu asigură comunicarea efectivă cu clienţii ci este responsabil cu "ascultarea" reţelei şi crearea unor obiecte de tip `Socket` pentru fiecare cerere apărută, prin intermediul căruia va fi realizată legătura cu clientul. Crearea unui obiect de tip `ServerSocket` se face specificând portul la care rulează serverul, constructorul folosit fiind:

```
ServerSocket(int port)
```

Metoda clasei `ServerSocket` care aşteaptă "ascultă" reţeaua este **accept**. Aceasta blochează procesul părinte până la apariţia unui cereri şi returnează un nou obiect de tip `Socket` ce va asigura comunicarea cu clientul. Blocarea poate să nu fie permanentă ci doar pentru o anumită perioadă de timp - aceasta va fi specificată prin metoda **setSoTimeout**, cu argumentul dat în milisecunde.



Pentru fiecare din cele două socketuri deschise pot fi create apoi două fluxuri pe octeţi pentru citirea, respectiv scrierea datelor. Acest lucru se realizează prin intermediul metodelor **getInputStream**, respectuv **getOutputStream**. Fluxurile obţinute vor fi folosite împreună cu fluxuri de procesare care să asigure o comunicare facilă între cele două procese. In funcţie de specificul aplicaţiei acestea pot fi perechile:

- `BufferedReader`, `BufferedWriter` şi `PrintWriter` - pentru comunicare prin intermediul şirurilor de caractere;
- `DataInputStream`, `DataOutputStream` - pentru comunicare prin date primitive;
- `ObjectInputStream`, `ObjectOutputStream` - pentru cominicare prin intermediul obiectelor;

Structura generală a unui server bazat pe conexiuni este:

```
1. Creeaza un obiect de tip ServerSocket la un anumit port
while (true) {
  2. Asteapta realizarea unei conexiuni cu un client,
     folosind metoda accept;
     (va fi creat un obiect nou de tip Socket)
  3. Trateaza cererea venita de la client:
     3.1 Deschide un flux de intrare si primeste cererea
     3.2 Deschide un flux de iesire si trimite raspunsul
     3.3 Inchide fluxurile si socketul nou creat
}
```

Este recomandat ca tratarea cererilor să se realizeze în fire de execuţie separate, pentru ca metoda `accept` să poată fi reapelată cât mai repede în vederea stabilirii conexiunii cu un alt client.

Structura generală a unui client bazat pe conexiuni este:

```
1. Citeste sau declara adresa IP a serverului si portul
   la care acesta ruleaza;
2. Creeaza un obiect de tip Socket cu adresa si portul specificate;
3. Comunica cu serverul:
   3.1 Deschide un flux de iesire si trimite cererea;
   3.2 Deschide un flux de intrare si primeste raspunsul;
   3.3 Inchide fluxurile si socketul creat;
```

In exemplul următor vom implementa o aplicaţie client-server folosind comunicarea prin conexiuni. Clientul va trimite serverului un nume iar acesta va raspunde prin mesajul "Hello *nume*". Tratarea cererilor se va face în fire de execuţie separate.

Listing 13.2: Structura unui server bazat pe conexiuni

```
import java.net.*;
import java.io.*;

class ClientThread extends Thread {
  Socket socket = null;

  public ClientThread(Socket socket) {
    this.socket = socket;
```

```
  }

  public void run() {
    //Executam solicitarea clientului
    String cerere, raspuns;
    try {
      // in este fluxul de intrare de la client
      BufferedReader in = new BufferedReader(new
          InputStreamReader(
        socket.getInputStream() ));

      // out este flux de iesire catre client
      PrintWriter out = new PrintWriter(
        socket.getOutputStream());

      // Primim cerere de la client
      cerere = in.readLine();

      // Trimitem raspuns clientului
      raspuns = "Hello " + cerere + "!";
      out.println(raspuns);
      out.flush();

    } catch (IOException e) {
      System.err.println("Eroare IO \n" + e);
    } finally {
      // Inchidem socketul deschis pentru clientul curent
      try {
        socket.close();
      } catch (IOException e) {
        System.err.println("Socketul nu poate fi inchis \n" +
            e);
      }
    }
  }
}

public class SimpleServer {

  // Definim portul pe care se gaseste serverul
  // (in afara intervalului 1-1024)
  public static final int PORT = 8100;

  public SimpleServer() throws IOException {
    ServerSocket serverSocket = null;
```

```
    try {
      serverSocket = new ServerSocket(PORT);
      while (true) {
        System.out.println("Asteptam un client...");
        Socket socket = serverSocket.accept();

        // Executam solicitarea clientului intr-un fir de
            executie
        ClientThread t = new ClientThread(socket);
        t.start();
      }
    } catch (IOException e) {
      System.err.println("Eroare IO \n" + e);
    } finally {
      serverSocket.close();
    }
  }

  public static void main(String[] args) throws IOException {
    SimpleServer server = new SimpleServer();
  }
}
```

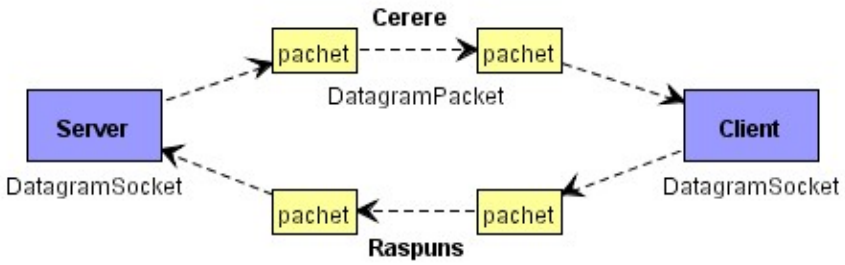
Listing 13.3: Structura unui client bazat pe conexiuni

```
import java.net.*;
import java.io.*;

public class SimpleClient {

  public static void main(String[] args) throws IOException {
    // Adresa IP a serverului
    String adresaServer = "127.0.0.1";

    // Portul la care serverul ofera serviciul
    int PORT = 8100;

    Socket socket = null;
    PrintWriter out = null;
    BufferedReader in = null;
    String cerere, raspuns;

    try {
      socket = new Socket(adresaServer, PORT);
```

```
      out = new PrintWriter(socket.getOutputStream(), true);
      in = new BufferedReader(new InputStreamReader(
         socket.getInputStream()));

      // Trimitem o cerere la server
      cerere = "Duke";
      out.println(cerere);

      //Asteaptam raspunsul de la server ("Hello Duke!")
      raspuns = in.readLine();
      System.out.println(raspuns);

    } catch (UnknownHostException e) {
      System.err.println("Serverul nu poate fi gasit \n" + e)
         ;
      System.exit(1);
    } finally {
      if (out != null)
        out.close();
      if (in != null)
        in.close();
      if (socket!= null)
        socket.close();
    }
  }
}
```

13.5 Comunicarea prin datagrame

In acest model nu există o conexiune permanentă între client şi server prin intermediul căreia să se realizeze comunicarea. Clientul trimite cererea către server prin intermediul unuia sau mai multor pachete de date independente, serverul le recepţionează, extrage informaţiile conţinute şi returnează răspunsul tot prin intermediul pachetelor. Un astfel de pachet se numeşte *datagramă* şi este reprezentat printr-un obiect din clasa **DatagramPacket**. Rutarea datagramelor de la o maşină la alta se face exclusiv pe baza informaţiilor conţinute de acestea. Primirea şi trimiterea datagramelor se realizează prin intermediul unui socket, modelat prin intermediul clasei **DatagramSocket**.



După cum am menţionat deja, dezavantajul acestei metode este că nu garantează ajungerea la destinaţie a pachetelor trimise şi nici că vor fi primite în aceeaşi ordinie în care au fost expediate. Pe de altă parte, există situaţii în care aceste lucruri nu sunt importante şi acest model este de preferat celui bazat pe conexiuni care solicită mult mai mult atât serverul cât şi clientul. De fapt, protocolul TCP/IP foloseşte tot pachete pentru trimiterea informaţiilor dintr-un nod în altul al reţelei, cu deosebirea că asigură respectarea ordinii de transmitere a mesajelor şi verifică ajungerea la destinaţie a tuturor pachetelor - în cazul în care unul nu a ajuns, acesta va fi retrimis automat.

Clasa `DatagramPacket` conţine următorii constructori:

```
DatagramPacket(byte[] buf, int length,
  InetAddress address, int port)
DatagramPacket(byte[] buf, int offset, int length,
  InetAddress address, int port)

DatagramPacket(byte[] buf, int offset, int length,
  SocketAddress address)
DatagramPacket(byte[] buf, int length,
  SocketAddress address)

DatagramPacket(byte[] buf, int length)
DatagramPacket(byte[] buf, int offset, int length)
```

Primele două perechi de constructori sunt pentru creare pachetelor ce vor fi **expediate**, diferenţa între ele fiind utilizarea claselor `InetAddress`, respectiv `SocketAddress` pentru specificarea adresei desinaţie.

A trei pereche de constructori este folosită pentru crearea unui pachet în care vor fi **recepţionate** date, ei nespecificând vreo sursă sau destinaţie.

După crearea unui pachet procesul de trimitere şi primire a acestuia implică apelul metodelor **send** şi **receive** ale clasei `DatagramSocket`. Deoarece toate informaţii sunt incluse în datagramă, acelaşi socket poate fi folosit atât pentru trimiterea de pachete, eventual către destinaţii diferite, cât şi pentru recepţionarea acestora de la diverse surse. In cazul în care refolosim pachete, putem schimba conţinutul acestora cu metoda **setData**, precum şi adresa la care le trimitem prin **setAddress, setPort** şi **setSocketAddress**.

Extragerea informaţiilor conţiunte de un pachet se realizează prin metoda **getData** din clasa `DatagramPacket`. De asemenea, această clasă oferă metode pentru aflarea adresei IP şi a portului procesului care a trimis datagrama, pentru a-i putea răspunde dacă este necesar. Acestea sunt: **getAdress, getPort** şi **getSocketAddress**.

Listing 13.4: Structura unui server bazat pe datagrame

```
import java.net.*;
import java.io.*;

public class DatagramServer {

  public static final int PORT = 8200;
  private DatagramSocket socket = null;
  DatagramPacket cerere, raspuns = null;

  public void start() throws IOException {

    socket = new DatagramSocket(PORT);
    try {
      while (true) {

        // Declaram pachetul in care va fi receptionata
           cererea
        byte[] buf = new byte[256];
        cerere = new DatagramPacket(buf, buf.length);

        System.out.println("Asteptam un pachet...");
        socket.receive(cerere);

        // Aflam adresa si portul de la care vine cererea
        InetAddress adresa = cerere.getAddress();
        int port = cerere.getPort();

        // Construim raspunsul
```

```
        String mesaj = "Hello " + new String(cerere.getData()
            );
        buf = mesaj.getBytes();

        // Trimitem un pachet cu raspunsul catre client
        raspuns = new DatagramPacket(buf, buf.length, adresa,
            port);
        socket.send(raspuns);
      }
    } finally {
      if (socket != null)
        socket.close();
    }
  }

  public static void main(String[] args) throws IOException {
    new DatagramServer().start();
  }
}
```

Listing 13.5: Structura unui client bazat pe datagrame

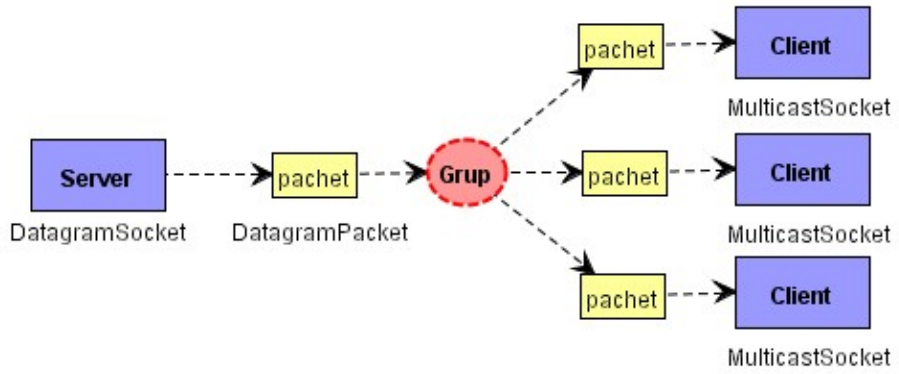
```
import java.net.*;
import java.io.*;

public class DatagramClient {

  public static void main(String[] args) throws IOException {

    // Adresa IP si portul la care ruleaza serverul
    InetAddress adresa = InetAddress.getByName("127.0.0.1");
    int port=8200;

    DatagramSocket socket = null;
    DatagramPacket packet = null;
    byte buf[];

    try {
      // Construim un socket pentru comunicare
      socket = new DatagramSocket();

      // Construim si trimitem pachetul cu cererea catre
          server
      buf = "Duke".getBytes();
```

```
      packet = new DatagramPacket(buf, buf.length, adresa,
         port);
      socket.send(packet);

      // Asteaptam pachetul cu raspunsul de la server
      buf = new byte[256];
      packet = new DatagramPacket(buf, buf.length);
      socket.receive(packet);

      // Afisam raspunsul ("Hello Duke!")
      System.out.println(new String(packet.getData()));
    } finally {
      if (socket != null)
        socket.close();
    }
  }
}
```

13.6 Trimiterea de mesaje către mai mulţi clienţi

Diverse situaţii impun gruparea mai multor clienţi astfel încât un mesaj (pachet) trimis pe adresa grupului să fie recepţionat de fiecare dintre aceştia. Gruparea mai multor programe în vederea trimiterii multiple de mesaje se realizează prin intermediul unui socket special, descris de clasa **MulticastSocket**, extensie a clasei `DatagramSocket`.

Un grup de clienţi abonaţi pentru trimitere multiplă este specificat printr-o adresă IP din intervalul `224.0.0.1 - 239.255.255.255` şi un port UDP. Adresa `224.0.0.0` este rezervată şi nu trebuie folosită.



Listing 13.6: Inregistrarea unui client într-un grup

```
import java.net.*;
import java.io.*;

public class MulticastClient {

  public static void main(String[] args) throws IOException {

    // Adresa IP si portul care reprezinta grupul de clienti
    InetAddress group = InetAddress.getByName("230.0.0.1");
    int port=4444;

    MulticastSocket socket = null;
    byte buf[];

    try {
      // Ne alaturam grupului aflat la adresa si portul
         specificate
      socket = new MulticastSocket(port);
      socket.joinGroup(group);

      // Asteaptam un pachet venit pe adresa grupului
      buf = new byte[256];
      DatagramPacket packet = new DatagramPacket(buf, buf.
         length);

      System.out.println("Asteptam un pachet...");
      socket.receive(packet);
```

```
      System.out.println(new String(packet.getData()).trim())
          ;
    } finally {
      if (socket != null) {
        socket.leaveGroup(group);
        socket.close();
      }
    }
  }
}
```

Listing 13.7: Transmiterea unui mesaj către un grup

```
import java.net.*;
import java.io.*;

public class MulticastSend {

  public static void main(String[] args) throws IOException {

    InetAddress grup = InetAddress.getByName("230.0.0.1");
    int port = 4444;
    byte[] buf;
    DatagramPacket packet = null;

    // Cream un socket cu un numar oarecare
    DatagramSocket socket = new DatagramSocket(0);
    try {
      // Trimitem un pachet catre toti clientii din grup
      buf = (new String("Salut grup!")).getBytes();
      packet = new DatagramPacket(buf, buf.length, grup, port
          );
      socket.send(packet);

    } finally {
      socket.close();
    }
  }
}
```

Capitolul 14

Appleturi

14.1 Introducere

Definiţie

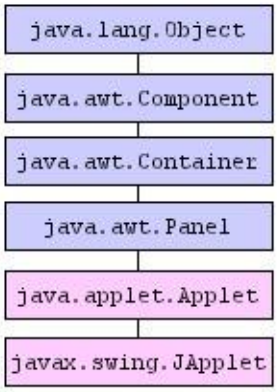
Un *applet* reprezintă un program Java de dimensiuni reduse ce gestionează o suprafaţă de afişare (container) care poate fi inclusă într-o pagină Web. Un astfel de program se mai numeşte *miniaplicatie*.

Ca orice altă aplicaţie Java, codul unui applet poate fi format din una sau mai multe clase. Una dintre acestea este *principală* şi extinde clasa `Applet`, aceasta fiind clasa ce trebuie specificată în documentul HTML ce descrie pagina Web în care dorim să includem appletul.

Diferenţa fundamentală dintre un applet şi o aplicaţie constă în faptul că un applet nu poate fi executat independent, ci va fi executat de browserul în care este încărcată pagina Web ce conţine appletul respectiv. O aplicaţie independentă este executată prin apelul interpretorului *java*, având ca argument numele clasei principale a aplicaţiei, clasa principală fiind cea care conţine metoda *main*. Ciclul de viată al unui applet este complet diferit, fiind dictat de evenimentele generate de către browser la vizualizarea documentului HTML ce conţine appletul.

Pachetul care oferă suport pentru crearea de appleturi este **java.applet**, cea mai importantă clasă fiind **Applet**. In pachetul **javax.swing** există şi clasa **JApplet**, care extinde `Applet`, oferind suport pentru crearea de appleturi pe arhitectura de componente JFC/Swing.

Ierarhia claselor din care derivă appleturile este prezentata în figura de mai jos:



Fiind derivată din clasa `Container`, clasa `Applet` descrie de fapt suprafeţe de afişare, asemenea claselor `Frame` sau `Panel`.

14.2 Crearea unui applet simplu

Crearea structurii de fişiere şi compilarea applet-urilor sunt identice ca în cazul aplicaţiilor. Diferă în schimb structura programului şi modul de rulare a acestuia. Să parguream în continuare aceşti paşi pentru a realiza un applet extrem de simplu, care afişează o imagine şi un şir de caractere.

1. Scrierea codului sursa

```
import java.awt.* ;
import java.applet.* ;

public class FirstApplet extends Applet {
  Image img;
  public void init() {
    img = getImage(getCodeBase(), "taz.gif");
  }
```

```
  public void paint (Graphics g) {
    g.drawImage(img, 0, 0, this);
    g.drawOval(100,0,150,50);
    g.drawString("Hello! My name is Taz!", 110, 25);
  }
}
```

Pentru a putea fi executată de browser, clasa principală a appletului trebuie să fie publică.

2. Salvarea fisierelor sursă

Ca orice clasă publică, clasa principala a appletului va fi salvată într-un fişier cu acelaşi nume şi extensia `.java`. Aşadar, vom salva clasa de mai sus într-un fişier `FirstApplet.java`.

3. Compilarea

Compilarea se face la fel ca şi la aplicaţiile independente, folosind compilatorul `javac` apelat pentru fişierul ce conţine appletul.

```
javac FirstApplet.java
```

In cazul în care compilarea a reuşit va fi generat fisierul `FirstApplet.class`.

4. Rularea appletului

Applet-urile nu ruleaza independent. Ele pot fi rulate doar prin intermediul unui browser: Internet Explorer, Netscape, Mozilla, Opera, etc. sau printr-un program special cum ar fi **appletviewer** din kitul de dezvoltare J2SDK. Pentru a executa un applet trebuie să facem două operaţii:

- **Crearea unui fişier HTML** în care vom include applet-ul. Să considerăm fişierul `simplu.html`, având conţinutul de mai jos:

```
<html>
<head>
  <title>Primul applet Java</title>
</head>
<body>
  <applet code=FirstApplet.class width=400 height=400>
  </applet>
</body>
</html>
```

- **Vizualizarea appletului**: se deschide fisierul `simplu.html` folosind unul din browser-ele amintite sau efectuând apelul:
`appletviewer simplu.html`.

14.3 Ciclul de viaţă al unui applet

Execuţia unui applet începe în momentul în care un browser afişează o pagină Web în care este inclus appletul respectiv şi poate trece prin mai multe etape. Fiecare etapă este strâns legată de un eveniment generat de către browser şi determină apelarea unei metode specifice din clasa ce implementează appletul.

- **Incărcarea în memorie**
Este creată o instanţa a clasei principale a appletului şi încarcată în memorie.

- **Iniţializarea**
Este apelată metoda `init` ce permite iniţializarea diverselor variabile, citirea unor parametri de intrare, etc.

- **Pornirea**
Este apelată metoda `start`

- **Execuţia propriu-zisă**
Constă în interacţiunea dintre utilizator şi componentele afişate pe suprafaţa appletului sau în executarea unui anumit cod într-un fir de execuţie. In unele situaţii întreaga execuţie a appletului se consumă la etapele de iniţializare şi pornire.

- **Oprirea temporară**
  In cazul în care utilizatorul părăseşte pagina Web în care rulează appletul este apelată metoda `stop` a acestuia, dându-i astfel posibilitatea să opreasca temporar execuţia sa pe perioada în care nu este vizibil, pentru a nu consuma inutil din timpul procesorului. Acelaşi lucru se întâmplă dacă fereastra browserului este minimizată. In momentul când pagina Web ce contine appletul devine din nou activă, va fi reapelată metoda `start`.

- **Oprirea definitivă**
  La închiderea tuturor instanţelor browserului folosit pentru vizualizare, appletul va fi eliminat din memorie şi va fi apelată metoda `destroy` a acestuia, pentru a-i permite să elibereze resursele deţinute. Apelul metodei `destroy` este întotdeauna precedat de apelul lui `stop`.

Metodele specifice appleturilor

Aşadar, există o serie de metode specifice appleturilor ce sunt apelate automat la diverse evenimente generate de către browser. Acestea sunt definite în clasa `Applet` şi sunt enumerate în tabelul de mai jos:

Metoda	**Situaţia în care este apelată**
init	La iniţializarea appletului. Teoretic, această metodă ar trebui să se apeleze o singură dată, la prima afişare a appletului în pagină, însă, la unele browsere, este posibil ca ea să se apeleze de mai multe ori.
start	Imediat după iniţializare şi de fiecare dată când appletul redevine activ, după o oprire temporară.
stop	De fiecare dată când appletul nu mai este vizibil (pagina Web nu mai este vizibilă, fereastra browserului este minimizată, etc) şi înainte de `destroy`.
destroy	La închiderea ultimei instanţe a browserului care a încărcat în memorie clasa principală a appletului.

Atenţie

Aceste metode sunt apelate automat de browser şi nu trebuie apelate explicit din program !

Structura generală a unui applet

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

public class StructuraApplet extends Applet {

  public void init() {
  }

  public void start() {
  }

  public void stop() {
  }

  public void destroy() {
  }

}
```

14.4 Interfaţa grafică cu utilizatorul

După cum am văzut, clasa `Applet` este o extensie a superclasei `Container`, ceea ce înseamnă că appleturile sunt, înainte de toate, suprafeţe de afişare. Plasarea componentelor, gestionarea poziţionării lor şi tratarea evenimentelor generate se realizează la fel ca şi în cazul aplicaţiilor. Uzual, adăugarea componentelor pe suprafaţa appletului precum şi stabilirea obiectelor responsabile cu tratarea evenimentelor generate sunt operaţiuni ce vor fi realizate în metoda `init`.

Gestionarul de poziţionare implicit este `FlowLayout`, însă acesta poate fi schimbat prin metoda `setLayout`.

Desenarea pe suprafaţa unui applet

Există o categorie întreagă de appleturi ce nu comunică cu utilizatorul prin intermediul componentelor ci, execuţia lor se rezumă la diverse operaţiuni de desenare realizate în metoda **paint**. Reamintim că metoda `paint` este responsabilă cu definirea aspectului grafic al oricărei componente. Implicit, metoda `paint` din clasa `Applet` nu realizează nimic, deci, în cazul în care dorim să desenăm direct pe suprafaţa unui applet va fi nevoie să supradefinim această metodă.

```
public void paint(Graphics g) {
  // Desenare
 ...
}
```

In cazul în care este aleasă această soluţie, evenimentele tratate uzual vor fi cele generate de mouse sau tastatură.

14.5 Definirea şi folosirea parametrilor

Parametrii sunt pentru appleturi ceea ce argumentele de la linia de comandă sunt pentru aplicaţiile independente. Ei permit utilizatorului să personalizeze aspectul sau comportarea unui applet fără a-i schimba codul şi recompila clasele.

Definirea parametrilor se face în cadrul tagului `APPLET` din documentul HTML ce conţine appletul şi sunt identificaţi prin atributul **PARAM**. Fiecare parametru are un nume, specificat prin **NAME** şi o valoare, specificată prin **VALUE**, ca în exemplul de mai jos:

```
<APPLET CODE="TestParametri.class" WIDTH=100 HEIGHT=50
  <PARAM NAME=textAfisat VALUE="Salut">
  <PARAM NAME=numeFont VALUE="Times New Roman">
  <PARAM NAME=dimFont VALUE=20>
</APPLET>
```

Ca şi în cazul argumentelor trimise aplicaţiilor de la linia de comandă, tipul parametrilor este întotdeauna **şir de caractere**, indiferent dacă valoarea este între ghilimele sau nu.

Fiecare applet are şi un set de parametri prestabiliţi ale căror nume nu vor putea fi folosite pentru definirea de noi parametri folosind metoda de

mai sus. Aceştia apar direct în corpul tagului `APPLET` şi definesc informaţii generale despre applet. Exemple de astfel de parametri sunt `CODE`, `WIDTH` sau `HEIGHT`. Lista lor completă va fi prezentata la descrierea tagului `APPLET`.

Folosirea parametrilor primiţi de către un applet se face prin intermediul metodei **getParameter** care primeşte ca argument numele unui parametru şi returnează valoarea acestuia. In cazul în care nu există nici un parametru cu numele specificat, metoda întoarce `null`, caz în care programul trebuie să atribuie o valoare implicită variabilei în care se dorea citirea respectivului parametru.

Orice applet poate pune la dispoziţie o "documentaţie" referitoare la parametrii pe care îi suportă, pentru a veni în ajutorul utilizatorilor care doresc să includă appletul într-o pagină Web. Aceasta se realizează prin supradefinirea metodei **getParameterInfo**, care returnează un vector format din triplete de şiruri. Fiecare element al vectorului este de fapt un vector cu trei elemente de tip `String`, cele trei şiruri reprezentând *numele* parametrului, *tipul* său şi o *descriere* a sa. Informaţiile furnizate de un applet pot fi citite din browserul folosit pentru vizualizare prin metode specifice acestuia. De exemplu, în appletviewer informaţiile despre parametri pot fi vizualizate la rubrica *Info* din meniul *Applet*, în Netscape se foloseşte opţiunea *Page info* din meniul *View*, etc.

Să scriem un applet care să afişeze un text primit ca parametru, folosind un font cu numele şi dimensiunea specificate de asemenea ca parametri.

Listing 14.1: Folosirea parametrilor

```
import java.applet.Applet;
import java.awt.*;

public class TestParametri extends Applet  {

  String text, numeFont;
  int dimFont;

  public void init() {
    text = getParameter("textAfisat");
    if (text == null)
      text = "Hello"; // valoare implicita

    numeFont = getParameter("numeFont");
    if (numeFont == null)
      numeFont = "Arial";
```

```
    try {
      dimFont = Integer.parseInt(getParameter("dimFont"));
    } catch(NumberFormatException e) {
      dimFont = 16;
    }
  }

  public void paint(Graphics g) {
    g.setFont(new Font(numeFont, Font.BOLD, dimFont));
    g.drawString(text, 20, 20);
  }

  public String[][] getParameterInfo() {
    String[][] info = {
      //  Nume           Tip           Descriere
      {"textAfisat", "String", "Sirul ce va fi afisat"},
      {"numeFont",   "String", "Numele fontului"},
      {"dimFont",    "int",    "Dimensiunea fontului"}
    };
    return info;
  }

}
```

14.6 Tag-ul APPLET

Sintaxa completă a tagului APPLET, cu ajutorul căruia pot fi incluse appleturi în cadrul paginilor Web este:

```
<APPLET
    CODE = clasaApplet
    WIDTH = latimeInPixeli
    HEIGHT = inaltimeInPixeli

    [ARCHIVE = arhiva.jar]
    [CODEBASE = URLApplet]
    [ALT = textAlternativ]
    [NAME = numeInstantaApplet]
    [ALIGN = aliniere]
    [VSPACE = spatiuVertical]
```

```
    [HSPACE = spatiuOrizontal] >

  [< PARAM NAME = parametru1 VALUE = valoare1 >]
  [< PARAM NAME = parametru2 VALUE = valoare2 >]
  ...
  [text HTML alternativ]

</APPLET>
```

Atributele puse între paranteze pătrate sunt opţionale.

- **CODE** = *clasaApplet*
  Numele fişierului ce conţine clasa principală a appletului. Acesta va fi căutat în directorul specificat de `CODEBASE`. Nu poate fi absolut şi trebuie obligatoriu specificat. Extensia ".class" poate sau nu să apară.

- **WIDTH** =*latimeInPixeli*, **HEIGHT** =*inaltimeInPixeli*
  Specifică lăţimea şi înălţimea suprafeţei în care va fi afişat appletul. Sunt obligatorii.

- **ARCHIVE** = *arhiva.jar*
  Specifică arhiva în care se găsesc clasele appletului.

- **CODEBASE** = *directorApplet*
  Specifică URL-ul la care se găseşte clasa appletului. Uzual se exprimă relativ la directorul documentului HTML. In cazul în care lipseşte, se consideră implicit URL-ul documentului.

- **ALT** = *textAlternativ*
  Specifică textul ce trebuie afişat dacă browserul înţelege tagul `APPLET` dar nu poate rula appleturi Java.

- **NAME** =*numeInstantaApplet*
  Oferă posibilitatea de a da un nume respectivei instanţe a appletului, astfel încât mai multe appleturi aflate pe aceeaşi pagină să poată comunica între ele folosindu-se de numele lor.

- **ALIGN** =*aliniere*
  Semnifică modalitatea de aliniere a appletului în pagina Web. Acest atribut poate primi una din următoarele valori: `left, right, top,`

`texttop, middle, absmiddle, baseline, bottom, absbottom` , seminificaţiile lor fiind aceleaşi ca şi la tagul `IMG`.

- **VSPACE** =*spatiuVertical*, **HSPACE** = *spatiuOrizontal*
  Specifică numarul de pixeli dintre applet şi marginile suprafetei de afişare.

- **PARAM**
  Tag-urile `PARAM` sunt folosite pentru specificarea parametrilor unui applet (vezi "Folosirea parametrilor").

- *text HTML alternativ*
  Este textul ce va fi afişat în cazul în care browserul nu întelege tagul `APPLET`. Browserele *Java-enabled* vor ignora acest text.

14.7 Folosirea firelor de execuţie în appleturi

La încărcarea unei pagini Web, fiecărui applet îi este creat automat un fir de execuţie responsabil cu apelarea metodelor acestuia. Acestea vor rula concurent după regulile de planificare implementate de maşina virtuală Java a platformei folosite.

Din punctul de vedere al interfeţei grafice însă, fiecare applet aflat pe o pagină Web are acces la un **acelaşi** fir de execuţie, creat de asemenea automat de către browser, şi care este responsabil cu desenarea appletului (apelul metodelor `update` şi `paint`) precum şi cu transmiterea mesajelor generate de către componente. Intrucât toate appleturile de pe pagină "împart" acest fir de execuţie, nici unul nu trebuie să îl solicite în mod excesiv, deoarece va provoca funcţionarea anormală sau chiar blocarea celorlalte.

In cazul în care dorim să efectuăm operaţiuni consumatoare de timp este recomandat să le realizăm într-un alt fir de execuţie, pentru a nu bloca interacţiunea utilizatorului cu appletul, redesenarea acestuia sau activitatea celorlalte appleturi de pe pagină.

Să considermăm mai întâi două abordări greşite de lucru cu appleturi. Dorim să creăm un applet care să afişeze la coordonate aleatoare mesajul "Hello", cu pauză de o secundă între două afişări. Prima variantă, greşită de altfel, ar fi:

Listing 14.2: Incorect: blocarea metodei `paint`

```
import java.applet.*;
import java.awt.*;

public class AppletRau1 extends Applet {
  public void paint(Graphics g) {
    while(true) {
      int x = (int)(Math.random() * getWidth());
      int y = (int)(Math.random() * getHeight());
      g.drawString("Hello", x, y);
      try {
        Thread.sleep(1000);
      } catch (InterruptedException e) {}
    }
  }
}
```

Motivul pentru care acest applet nu funcţionează corect şi probabil va duce la anomalii în funcţionarea browserului este că firul de execuţie care se ocupă cu desenarea va rămâne blocat în metoda `paint`, încercând să o termine. Ca regulă generală, codul metodei `paint` trebuie să fie cât mai simplu de executat ceea ce, evident, nu este cazul în appletul de mai sus.

O altă idee de rezolvare care ne-ar putea veni, de asemenea greşită, este următoarea :

Listing 14.3: Incorect: appletul nu termină iniţializarea

```
import java.applet.*;
import java.awt.*;

public class AppletRau2 extends Applet {
  int x, y;

  public void init() {
    while(true) {
      x = (int)(Math.random() * getWidth());
      y = (int)(Math.random() * getHeight());
      repaint();
      try {
        Thread.sleep(1000);
      } catch (InterruptedException e) {}
    }
  }
  public void paint(Graphics g) {
```

```
    g.drawString("Hello", x, y);
  }
}
```

Pentru a putea da o soluţie corectă problemei propuse, trebuie să folosim un fir de execuţie propriu. Structura unui applet care doreste să lanseze un fir de execuţie poate avea două forme. In prima situaţie appletul porneşte firul la iniţialzarea sa iar acesta va rula, indiferent dacă appletul mai este sau nu vizibil, până la oprirea sa naturală (terminarea metodei `run`) sau până la închiderea sesiunii de lucru a browserului.

Listing 14.4: Corect: folosirea unui fir de execuţie propriu

```
import java.applet.*;
import java.awt.*;

public class AppletCorect1 extends Applet implements Runnable
    {
  int x, y;
  Thread fir = null;

  public void init() {
    if (fir == null) {
      fir = new Thread(this);
      fir.start();
    }
  }

  public void run() {
    while(true) {
      x = (int)(Math.random() * getWidth());
      y = (int)(Math.random() * getHeight());
      repaint();
      try {
        Thread.sleep(1000);
      } catch (InterruptedException e) {}
    }
  }
  public void paint(Graphics g) {
    g.drawString("Hello", x, y);
  }
}
```

In cazul în care firul de execuţie pornit de applet efectuează operatii ce

au sens doar dacă appletul este vizibil, cum ar fi animatie, ar fi de dorit ca acesta să se opreasca atunci când appletul nu mai este vizibil (la apelul metodei `stop`) şi să reporneasca atunci când appletul redevine vizibil (la apelul metodei `start`). Un applet este considerat *activ* imediat după apelul metodei `start` şi devine inactiv la apelul metodei `stop`. Pentru a afla dacă un applet este activ se foloseşte metoda **isActive**.

Să modificăm programul anterior, adăugând şi un contor care să numere afişările de mesaje - acesta nu va fi incrementat pe perioada în care appletul nu este activ.

Listing 14.5: Folosirea metodelor `start` şi `stop`

```
import java.applet.*;
import java.awt.*;

public class AppletCorect2 extends Applet implements Runnable
    {
  int x, y;
  Thread fir = null;
  boolean activ = false;
  int n = 0;

  public void start() {
    if (fir == null) {
      fir = new Thread(this);
      activ = true;
      fir.start();
    }
  }

  public void stop() {
    activ = false;
    fir = null;
  }

  public void run() {
    while(activ) {
      x = (int)(Math.random() * getWidth());
      y = (int)(Math.random() * getHeight());
      n ++;
      repaint();
      try {
        Thread.sleep(1000);
      } catch (InterruptedException e) {}
```

```
    }
  }

  public void paint(Graphics g) {
    g.drawString("Hello " + n, x, y);
  }
}
```

Atenţie

Este posibil ca unele browsere să nu apele metoda `stop` în situaţiile prevăzute în specficiaţiile appleturilor. Din acest motiv, corectitudinea unui applet nu trebuie să se bazeze pa acest mecanism.

14.8 Alte metode oferite de clasa Applet

Pe lângă metodele de bază: `init, start, stop, destroy`, clasa `Applet` oferă metode specifice applet-urilor cum ar fi:

Punerea la dispozitie a unor informaţii despre applet

Similară cu metoda `getParameterInfo` ce oferea o "documentaţie" despre parametrii pe care îi acceptă un applet, există metoda **getAppletInfo** ce permite specificarea unor informaţii legate de applet cum ar fi numele, autorul, versiunea, etc. Metoda returnează un sir de caractere continând informaţiile respective.

```
public String getAppletInfo() {
  return "Applet simplist, autor necunoscut, ver 1.0";
}
```

Aflarea adreselor URL referitoare la applet

Se realizează cu metodele:

- **getCodeBase** - ce returnează URL-ul directorului ce conţine clasa appletului;
- **getDocumentBase** - returnează URL-ul directorului ce conţine documentul HTML în care este inclus appletul respectiv.

Aceste metode sunt foarte utile deoarece permit specificarea relativă a unor fişiere folosite de un applet, cum ar fi imagini sau sunete.

Afişarea unor mesaje în bara de stare a browserului

Acest lucru se realizează cu metoda **showStatus**

```
public void init() {
  showStatus("Initializare applet...");
}
```

Afişarea imaginilor

Afişarea imaginilor într-un applet se face fie prin intermediul unei componente ce permite acest lucru, cum ar fi o suprafaţă de desenare de tip `Canvas`, fie direct în metoda `paint` a applet-ului, folosind metoda `drawImage` a clasei `Graphics`. In ambele cazuri, obţinerea unei referinţe la imaginea respectivă se va face cu ajutorul metodei **getImage** din clasa `Applet`. Aceasta poate primi ca argument fie adresa URL absolută a fişierului ce reprezintă imaginea, fie calea relativă la o anumită adresă URL, cum ar fi cea a directorului în care se găseşte documentul HTML ce conţine appletul (`getDocumentBase`) sau a directorului în care se găseşte clasa appletului (`getCodeBase`).

Listing 14.6: Afişarea imaginilor

```
import java.applet.Applet;
import java.awt.*;

public class Imagini extends Applet  {
  Image img = null;

  public void init() {
    img = getImage(getCodeBase(), "taz.gif");
  }
```

```
  public void paint(Graphics g) {
    g.drawImage(img, 0, 0, this);
  }
}
```

Aflarea contextului de execuţie

Contextul de execuţie al unui applet se referă la pagina în care acesta rulează, eventual împreună cu alte appleturi, şi este descris de interfaţa **AppletContext**. Crearea unui obiect ce implementează această interfaţă se realizează de către browser, la apelul metodei **getAppletContext** a clasei `Applet`. Prin intermediul acestei interfeţe un applet poate "vedea" în jurul sau, putând comunica cu alte applet-uri aflate pe aceeasi pagină sau cere browser-ului să deschidă diverse documente.

```
AppletContext contex = getAppletContext();
```

Afişarea unor documente în browser

Se face cu metoda **showDocument** ce primeşte adresa URL a fişierului ce conţine documentul pe care dorim sa-l deschidem (text, html, imagine, etc). Această metodă este accesată prin intermediul contextului de execuţie al appletului.

```
try {
  URL doc = new URL("http://www.infoiasi.ro");
  getAppletContext().showDocument(doc);
} catch(MalformedURLException e) {
  System.err.println("URL invalid! \n" + e);
}
```

Comunicarea cu alte applet-uri

Această comunicare implică de fapt identificarea unui applet aflat pe aceeaşi pagina şi apelarea unei metode sau setarea unei variabile publice a acestuia. Identificarea se face prin intermediu numelui pe care orice instanţa a unui applet îl poate specifica prin atributul `NAME`.

Obţinerea unei referinţe la un applet al cărui nume îl cunoaştem sau obţinerea unei enumerări a tuturor applet-urilor din pagină se fac prin intermediul contextului de execuţie, folosind metodele **getApplet**, respectiv **getApplets**.

Redarea sunetelor

Clasa `Applet` oferă şi posibilitatea redării de sunete în format **.au**. Acestea sunt descrise prin intermediul unor obiecte ce implementează interfaţa **AudioClip** din pachetul `java.applet`. Pentru a reda un sunet aflat într-un fişier ".au" la un anumit URL există două posibilităţi:

- Folosirea metodei **play** din clasa `Applet` care primeşte ca argument URL-ul la care se află sunetul; acesta poate fi specificat absolut sau relativ la URL-ul appletului
- Crearea unui obiect de tip `AudioClip` cu metoda **getAudioClip** apoi apelarea metodelor **start**, **loop** şi **stop** pentru acesta.

Listing 14.7: Redarea sunetelor

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

public class Sunete extends Applet implements ActionListener{
  Button play = new Button("Play");
  Button loop = new Button("Loop");
  Button stop = new Button("Stop");
  AudioClip clip = null;

  public void init() {
    // Fisierul cu sunetul trebuie sa fie in acelasi
    // director cu appletul
    clip = getAudioClip(getCodeBase(), "sunet.au");
    add(play);
    add(loop);
    add(stop);

    play.addActionListener(this);
    loop.addActionListener(this);
    stop.addActionListener(this);
```

```
  }

  public void actionPerformed(ActionEvent e) {
    Object src = e.getSource();
    if (src == play)
      clip.play();
    else if (src == loop)
      clip.loop();
    else if (src == stop)
      clip.stop();
  }
}
```

In cazul în care appletul foloseşte mai multe tipuri de sunete, este recomandat ca încărcarea acestora să fie făcută într-un fir de execuţie separat, pentru a nu bloca temporar activitatea firească a programului.

14.9 Arhivarea appleturilor

După cum am văzut, pentru ca un applet aflat pe o pagină Web să poată fi executat codul său va fi transferat de pe serverul care găzduieşte pagina Web solicitată pe maşina clientului. Deoarece transferul datelor prin reţea este un proces lent, cu cât dimensiunea fişierlor care formează appletul este mai redusă, cu atâ încărcarea acestuia se va face mai repede. Mai mult, dacă appletul conţine şi alte clase în afară de cea principală sau diverse resurse (imagini, sunete, etc), acestea vor fi transferate prin reţea abia în momentul în care va fi nevoie de ele, oprind temporar activitatea appletului până la încărcarea lor. Din aceste motive, cea mai eficientă modalitate de a distribui un applet este să arhivăm toate fişierele necesare acestuia.

Arhivarea fişierelor unui applet se face cu utilitarul **jar**, oferit în distribuţia J2SDK.

```
// Exemplu
jar cvf arhiva.jar ClasaPrincipala.class AltaClasa.class
        imagine.jpg sunet.au
// sau
jar cvf arhiva.jar *.class *.jpg *.au
```

Includerea unui applet arhivat într-o pagină Web se realizează specificând pe lângă numele clasei principale şi numele arhivei care o conţine:

```
<applet archive=arhiva.jar code=ClasaPrincipala
        width=400 height=200 />
```

14.10 Restricţii de securitate

Deoarece un applet se execută pe maşina utilizatorului care a solicitat pagina Web ce conţine appletul respectiv, este foarte important să existe anumite restricţii de securitate care să controleze activitatea acestuia, pentru a preveni acţiuni rău intenţionate, cum ar fi ştergeri de fişiere, etc., care să aducă prejudicii utilizatorului. Pentru a realiza acest lucru, procesul care rulează appleturi instalează un manager de securitate, adică un obiect de tip `SecurityManager` care va "superviza" activitatea metodelor appletului, aruncând excepţii de tip `Security Exception` în cazul în care una din acestea încearcă să efectueze o operaţie nepermisă.

Un applet nu poate să:

- Citească sau să scrie fişiere pe calculatorul pe care a fost încarcat (client).
- Deschidă conexiuni cu alte maşini în afară de cea de pe care provine (host).
- Pornească programe pe maşina client.
- Citească diverse proprietăţi ale sistemului de operare al clientului.

Ferestrele folosite de un applet, altele decât cea a browserului, vor arăta altfel decât într-o aplicaţie obişnuită, indicând faptul că au fost create de un applet.

14.11 Appleturi care sunt şi aplicaţii

Deoarece clasa `Applet` este derivată din `Container`, deci şi din `Component`, ea descrie o suprafaţă de afişare care poate fi inclusă ca orice altă componentă într-un alt container, cum ar fi o fereastră. Un applet poate funcţiona şi ca o aplicaţie independentă astfel:

- Adăugăm metoda `main` clasei care descrie appletul, în care vom face operaţiunile următoare.
- Creăm o instanţă a appletului şi o adăugăm pe suprafaţa unei ferestre.
- Apelăm metodele `init` şi `start`, care ar fi fost apelate automat de către browser.
- Facem fereastra vizibilă.

Listing 14.8: Applet şi aplicaţie

```
import java.applet.Applet;
import java.awt.*;

public class AppletAplicatie extends Applet  {

  public void init() {
    add(new Label("Applet si aplicatie"));
  }

  public static void main(String args[]) {
    AppletAplicatie applet = new AppletAplicatie();
    Frame f = new Frame("Applet si aplicatie");
    f.setSize(200, 200);

    f.add(applet, BorderLayout.CENTER);
    applet.init();
    applet.start();

    f.show();
  }
}
```

Capitolul 15

Lucrul cu baze de date

15.1 Introducere

15.1.1 Generalităţi despre baze de date

Aplicaţiile care folosesc baze de date sunt, în general, aplicaţii complexe folosite pentru gestionarea unor informaţii de dimensiuni mari într-o manieră sigură şi eficientă.

Ce este o bază de date ?
La nivelul cel mai general, o *bază de date* reprezintă o modalitate de stocare a unor informaţii (date) pe un suport extern, cu posibilitatea regăsirii acestora. Uzual, o bază de date este memorată într-unul sau mai multe fişiere.

Modelul clasic de baze de date este cel *relaţional*, în care datele sunt memorate în tabele. Un tabel reprezintă o structură de date formată dintr-o mulţime de articole, fiecare articol având definite o serie de atribute - aceste atribute corespund coloanelor tabelului, în timp ce o linie va reprezenta un articol. Pe lânga tabele, o bază de date mai poate conţine: proceduri şi funcţii, utilizatori şi grupuri de utilizatori, tipuri de date, obiecte, etc.

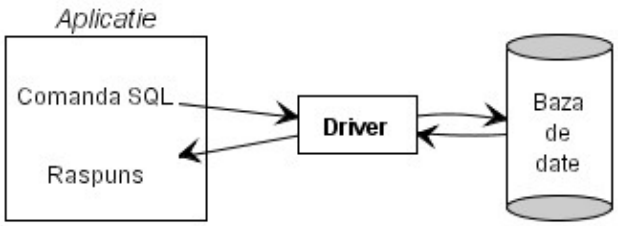
Dintre producătorii cei mai importanţi de baze de date amintim companiile Oracle, Sybase, IBM, Informix, Microsoft, etc. fiecare furnizând o serie întreagă de produse şi utilitare pentru lucrul cu baze de date. Aceste produse sunt în general referite prin termenii *DBMS* (Database Management System) sau, în traducere, *SGBD* (Sistem de Gestiune a Bazelor de Date). In acest capitol vom analiza lucrul cu baze de date din perspectiva programării

în limbajul Java, fără a descrie particularităţi ale unei soluţii de stocare a datelor anume. Vom vedea că, folosind Java, putem crea aplicaţii care să ruleze fără nici o modificare folosind diverse tipuri de baze care au aceeaşi structură, ducând în felul acesta noţiunea de portabilitate şi mai departe.

Crearea unei baze de date
Crearea unei baze de date se face uzual folosind aplicaţii specializate oferite de producătorul tipului respectiv de sistem de gestiune a datelor, dar există şi posibilitatea de a crea o baza folosind un script SQL. Acest aspect ne va preocupa însă mai puţin, exemplele prezentate presupunând că baza a fost creată deja şi are o anumită structură specificată.

Accesul la baza de date
Se face prin intermediul unui **driver** specific tipului respectiv de SGBD. Acesta este responsabil cu accesul efectiv la datele stocate, fiind legatura dintre aplicaţie şi baza de date.

Limbajul SQL
SQL (Structured Query Language) reprezintă un limaj de programare ce permite interogarea şi actualizarea informaţiilor din baze de date relaţionale. Acesta este standardizat astfel încât diverse tipuri de drivere să se comporte identic, oferind astfel o modalitate unitară de lucru cu baze de date.



15.1.2 JDBC

JDBC (Java Database Connectivity) este o interfaţă standard SQL de acces la baze de date. JDBC este constituită dintr-un set de clase şi interfeţe

scrise în Java, furnizând mecanisme standard pentru proiectanţii aplicaţiilor ce folosesc de baze de date.

Folosind JDBC este uşor să transmitem secvenţe SQL către baze de date relaţionale. Cu alte cuvinte, nu este necesar să scriem un program pentru a accesa o bază de date Oracle, alt program pentru a accesa o bază de date Sybase şi asa mai departe. Este de ajuns să scriem un singur program folosind API-ul JDBC şi acesta va fi capabil să comunice cu drivere diferite, trimiţând secvenţe SQL către baza de date dorită. Bineînţeles, scriind codul sursă în Java, ne este asigurată portabilitatea programului. Deci, iată două motive puternice care fac combinaţia Java - JDBC demnă de luat în seamă.

Pachetele care oferă suport pentru lucrul cu baze de date sunt **java.sql** ce reprezintă nucleul tehnologiei JDBC şi, preluat de pe platforma J2EE, **javax.sql**.

In linii mari, API-ul JDBC oferă următoarele facilităţi:

1. Stabilirea unei conexiuni cu o bază de date.
2. Efectuarea de secvenţe SQL.
3. Prelucrarea rezultatelor obţinute.

15.2 Conectarea la o bază de date

Procesul de conectare la o bază de date implică efectuarea a două operaţii:

1. Inregistrarea unui driver corespunzător.
2. Realizarea unei conexiuni propriu-zise.

Definiţie
O *conexiune (sesiune)* la o bază de date reprezintă un context prin care sunt trimise secvenţe SQL şi primite rezultate. Intr-o aplicaţie pot exista simultan mai multe conexiuni la baze de date diferite sau la aceeaşi bază.

Clasele şi interfeţele responsabile cu realizarea unei conexiuni sunt:

- **DriverManager** - este clasa ce se ocupă cu înregistrarea driverelor ce vor fi folosite în aplicaţie;

- **Driver** - interfaţa pe care trebuie să o implementeze orice clasă ce descrie un driver;
- **DriverPropertyInfo** - prin intermediul acestei clase pot fi specificate diverse proprietăţi ce vor fi folosite la realizarea conexiunilor;
- **Connection** - descrie obiectele ce modelează o conexiune propriu-zisă cu baza de date.

15.2.1 Inregistrarea unui driver

Primul lucru pe care trebuie să-l facă o aplicaţie în procesul de conectare la o bază de date este să înregistreze la maşina virtuală ce rulează aplicaţia driverul JDBC responsabil cu comunicarea cu respectiva bază de date. Acest lucru presupune încărcarea în memorie a clasei ce implementează driver-ul şi poate fi realizată în mai multe modalităţi.

a. Folosirea clasei **DriverManager**:
`DriverManager.registerDriver(new TipDriver());`

b. Folosirea metodei **Class.forName** ce apelează ClassLoader-ul maşinii virtuale:
`Class.forName("TipDriver");`
`Class.forName("TipDriver").newInstance();`

c. Setarea proprietăţii sistem **jdbc.drivers**, care poate fi realizată în două feluri:

   – De la linia de comandă:
   `java -Djdbc.drivers=TipDriver Aplicatie`
   – Din program:
   `System.setProperty("jdbc.drivers", "TipDriver");`

   Folosind această metodă, specificarea mai multor drivere se face separând numele claselor cu punct şi virgulă.

Dacă sunt înregistrate mai multe drivere, ordinea de precedenţă în alegerea driverului folosit la crearea unei noi conexiuni este:

1) Driverele înregistrate folosind proprietatea `jdbc.drivers` la iniţializarea maşinii virtuale ce va rula procesul.

2) Driverele înregistrate dinamic din aplicaţie.

15.2.2 Specificarea unei baze de date

O dată ce un driver JDBC a fost înregistrat, acesta poate fi folosit la stabilirea unei conexiuni cu o bază de date. Având în vedere faptul ca pot exista mai multe drivere încărcate în memorie, trebuie să avem posibilitea de a specifica pe lângă un identificator al bazei de date şi driverul ce trebuie folosit. Aceasta se realizează prin intermediul unei adrese specifice, numită **JDBC URL**, ce are următorul format:

jdbc:sub-protocol:identificator

Câmpul *sub-protocol* denumeşte tipul de driver ce trebuie folosit pentru realizarea conexiunii şi poate fi *odbc, oracle, sybase, db2* şi aşa mai departe.

Identificatorul bazei de date este un indicator specific fiecărui driver corespunzător bazei de date cu care aplicaţia doreşte să interacţioneze. In funcţie de tipul driver-ului acest identificator poate include numele unei maşini gazdă, un număr de port, numele unui fişier sau al unui director, etc., ca în exemplele de mai jos:

```
jdbc:odbc:test
jdbc:oracle:thin@persistentjava.com:1521:test
jdbc:sybase:test
jdbc:db2:test
```

Subprotocolul odbc este un caz specical, în sensul că permite specificarea în cadrul URL-ului a unor atribute ce vor fi realizate la crearea unei conexiuni. Sintaxa completa subprotocolului odbc este:

jdbc:odbc:identificator[;atribut=valoare]*

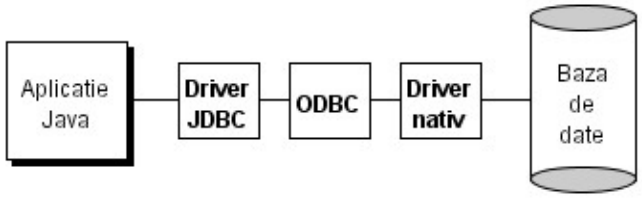
```
jdbc:odbc:test
jdbc:odbc:test;CacheSize=20;ExtensionCase=LOWER
jdbc:odbc:test;UID=duke;PWD=java
```

La primirea unui JDBC URL, `DriverManager`-ul va parcurge lista driverelor înregistrate în memorie, pâna când unul dintre ele va recunoaşte URL-ul respectiv. Dacă nu exista nici unul potrivit, atunci va fi lansata o excepţie de tipul `SQLException`, cu mesajul `"no suitable driver"`.

15.2.3 Tipuri de drivere

Tipurile de drivere existente ce pot fi folosite pentru realizarea unei conexiuni prin intermediul JDBC se împart în următoarele categorii:

Tip 1. *JDBC-ODBC Bridge*



Acest tip de driver permite conectarea la o bază de date care a fost înregistrată în prealabil în ODBC. ODBC (Open Database Conectivity) reprezintă o modalitate de a uniformiza accesul la baze de date, asociind acestora un identificator DSN (Data Source Name) şi diverşi parametri necesari conectării. Conectarea efectivă la baza de date se va face prin intermediul acestui identificator, driver-ul ODBC efectuând comunicarea cu driverul nativ al bazei de date.

Deşi simplu de utilizat, soluţia JDBC-ODBC nu este portabilă şi comunicarea cu baza de date suferă la nivelul vitezei de execuţie datorită multiplelor redirectări între drivere. De asemenea, atât ODBC-ul cât şi driver-ul nativ trebuie să existe pe maşina pe care rulează aplicaţia.

Clasa Java care descrie acest tip de driver JDBC este:

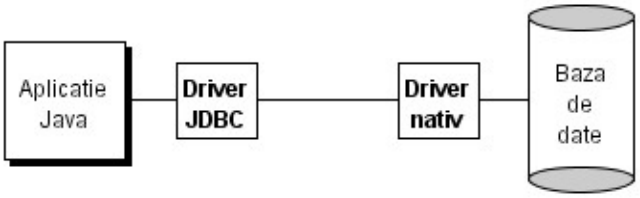
sun.jdbc.odbc.JdbcOdbcDriver

şi este inclusă în distribuţia standard J2SDK. Specificarea bazei de date se face printr-un URL de forma:

jdbc:odbc:identificator

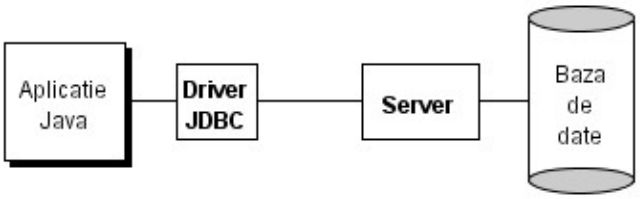
unde $identificator$ este profilul (DSN) creat bazei de date în ODBC.

Tip 2. *Driver JDBC - Driver nativ*



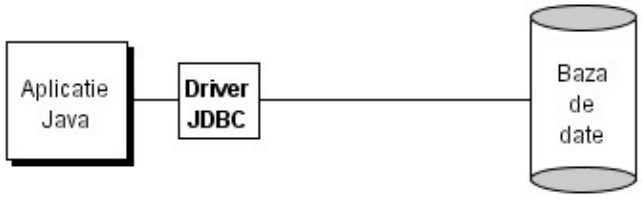
Acest tip de driver transformă cererile JDBC direct în apeluri către driverul nativ al bazei de date, care trebuie instalat în prealabil. Clase Java care implementează astfel de drivere pot fi procurate de la producătorii de SGBD-uri, distribuţia standard J2SDK neincluzând nici unul.

Tip 3. *Driver JDBC - Server*



Acest tip de driver transformă cererile JDBC folosind un protocol de reţea independent, acestea fiind apoi transormate folosind o aplicaţie server într-un protocol specfic bazei de date. Introducerea serverului ca nivel intermediar aduce flexibilitate maximă în sensul că vor putea fi realizate conexiuni cu diferite tipuri de baze, fără nici o modificare la nivelul clientului. Protocolul folosit este specific fiecărui producător.

Tip 4. *Driver JDBC nativ*



Acest tip de driver transformă cererile JDBC direct în cereri către baza de date folosind protocolul de reţea al acesteia. Această soluţie este cea mai rapidă, fiind preferată la dezvoltarea aplicaţiilor care manevrează volume mari de date şi viteza de execuţie este critică. Drivere de acest tip pot fi procurate de la diverşi producători de SGBD-uri.

15.2.4 Realizarea unei conexiuni

Metoda folosită pentru realizarea unei conexiuni este **getConnection** din clasa `DriverManager` şi poate avea mai multe forme:

```
Connection c = DriverManager.getConnection(url);
Connection c = DriverManager.getConnection(url, username, password);
Connection c = DriverManager.getConnection(url, dbproperties);
```

Stabilirea unei conexiuni folosind driverul JDBC-ODBC

```
String url = "jdbc:odbc:test" ;
// sau url = "jdbc:odbc:test;UID=duke;PWD=java" ;
try {
  Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
} catch(ClassNotFoundException e) {
  System.err.print("ClassNotFoundException: " + e) ;
  return ;
}

Connection con ;
try {
  con = DriverManager.getConnection(url, "duke", "java");
} catch(SQLException e) {
  System.err.println("SQLException: " + e);
```

```
} finally {
  try{
    con.close ;
  } catch(SQLException e) {
    System.err.println(SQLException: " + e) ;
  }
}
```

Stabilirea unei conexiuni folosind un driver MySql
Folosirea diferitelor tipuri de drivere implică doar schimbarea numelui clasei ce reprezintă driverul şi a modalităţii de specificare a bazei de date.

```
String url = "jdbc:mysql://localhost/test" ;
// sau url = "jdbc:mysql://localhost/test?user=duke&password=java";
try {
  Class.forName("com.mysql.jdbc.Driver") ;
} catch(ClassNotFoundException e) {
...
```

O conexiune va fi folosită pentru:

- Crearea de secvenţe SQL utilizate pentru interogarea sau actualizarea bazei.
- Aflarea unor informaţii legate de baza de date (meta-date).

De asemenea, clasa `Connection` asigură facilităţi pentru controlul tranzacţiilor din memorie către baza de date prin metodele **commit, rollback, setAutoCommit**.

Inchiderea unei conexiuni se realizează prin metoda **close**.

15.3 Efectuarea de secvenţe SQL

O dată facută conectarea cu metoda `DriverManager.getConection`, se poate folosi obiectul `Connection` rezultat pentru a se crea obiecte de tip **Statement**,**PreparedStatement** sau **CallableStatement** cu ajutorul cărora putem trimite secvenţe SQL către baza de date. Cele mai uzuale comenzi SQL sunt cele folosite pentru:

- Interogarea bazei de date: `SELECT`

- Actualizarea datelor: `INSERT, UPDATE, DELETE`
- Actualizarea structurii: `CREATE, ALTER, DROP` - acestea mai sunt numite instrucţiuni DDL (Data Definition Language)
- Apelarea unei proceduri stocate: `CALL`

După cum vom vedea, obţinerea şi prelucrarea rezultatelor unei interogări este realizată prin intermediul obiectelor de tip **ResultSet**.

15.3.1 Interfaţa Statement

Interfaţa `Statement` oferă metodele de bază pentru trimiterea de secvenţe SQL către baza de date şi obţinerea rezultatelor, celelalte două interfeţe: `PreparedStatement` şi `CallableStatement` fiind derivate din aceasta.

Crearea unui obiect `Statement` se realizează prin intermediul metodei **createStatement** a clasei `Connection`, fără nici un argument:

```
Connection con = DriverManager.getConnection(url);
Statement stmt = con.createStatement();
```

Execuţia unei secvenţe SQL poate fi realizată prin intermediul a trei metode:

1. executeQuery

Este folosită pentru realizarea de interogări de tip `SELECT`. Metoda returnează un obiect de tip `ResultSet` ce va conţine sub o formă tabelară rezultatul interogării.

```
String sql = "SELECT * FROM persoane";
ResultSet rs = stmt.executeQuery(sql);
```

2. executeUpdate

Este folosită pentru actualizarea datelor (`INSERT, UPDATE, DELETE`) sau a structurii bazei de date (`CREATE, ALTER, DROP`). Metoda va returna un întreg ce semnifică numărul de linii afectate de operaţiunea de actualizare a datelor, sau 0 în cazul unei instrucţiuni DDL.

```
String sql = "DELETE FROM persoane WHERE cod > 100";
int linii = stmt.executeUpdate(sql);
// Nr de articole care au fost afectate (sterse)

sql = "DROP TABLE temp";
stmt.executeUpdate(sql); // returneaza 0
```

3. execute

Această metodă va fi folosită doar dacâ este posibil ca rezultatul unei interogări să fie format din două sau mai multe obiecte de tip `ResultSet` sau rezultatul unei actualizări să fie format din mai mule valori, sau o combinaţie între aceste cazuri. Această situaţie, deşi mai rară, este posibilă atunci când sunt executate proceduri stocate sau secvenţe SQL cunoscute abia la momentul execuţiei, programatorul neştiind deci dacă va fi vorba de o actualizare a datelor sau a structurii. Metoda întoarce `true` dacă rezultatul obţinut este format din obiecte de tip `ResultSet` şi `false` dacă e format din întregi. In funcţie de aceasta, pot fi apelate metodele: **getResultSet** sau **getUpdateCount** pentru a afla efectiv rezultatul comenzii SQL. Pentru a prelua toate rezultatele va fi apelată metoda **getMoreResults**, după care vor fi apelate din nou metodele amintite, până la obţinerea valorii *null*, respectiv -1. Secvenţa completă de tratare a metodei `execute` este prezentată mai jos:

```
String sql = "comanda SQL necunoscuta";
stmt.execute(sql);
while(true) {
  int rowCount = stmt.getUpdateCount();
  if(rowCount > 0) {
    // Este o actualizare datelor
    System.out.println("Linii afectate = " + rowCount);
    stmt.getMoreResults();
    continue;
  }

  if(rowCount = 0) {
    // Comanda DDL sau nici o linie afectata
    System.out.println("Comanda DDL sau 0 actualizari");
```

```
      stmt.getMoreResults();
      continue;
    }

    // rowCount este -1
    // Avem unul sau mai multe ResultSet-uri
    ResultSet rs = stmt.getResultSet();
    if(rs != null) {
      // Proceseaza rezultatul
      ...
      stmt.getMoreResults();
      continue;
    }

    // Nu mai avem nici un rezultat
    break;
  }
```

Folosind clasa `Statement`, în cazul în care dorim să introducem valorile unor variabile într-o secvenţă SQL, nu avem altă soluţie decât să creăm un şir de caractere compus din instrucţiuni SQL şi valorile variabilelor:

```
int cod = 100;
String nume = "Popescu";
String sql = "SELECT * FROM persoane WHERE cod=" + cod +
 " OR nume='" + nume + "'";
ResultSet rs = stmt.executeQuery(sql);
```

15.3.2 Interfaţa PreparedStatement

Interfaţa PreparedStatement este derivată din `Statement`, fiind diferită de aceasta în următoarele privinţe:

- Instanţele de tip `PreparedStatement` conţin secvenţe SQL care au fost deja compilate (sunt "pregătite").
- O secvenţă SQL specificată unui obiect `PreparedStatement` poate să aibă unul sau mai mulţi parametri de intrare, care vor fi specificaţi prin intermediul unui semn de întrebare (**"?"**) în locul fiecăruia dintre

ei. Inainte ca secvenţa SQL să poată fi executată fiecărui parametru de intrare trebuie să i se atribuie o valoare, folosind metode specifice acestei clase.

Execuţia repetată a aceleiaşi secvenţe SQL, dar cu parametri diferiţi, va fi în general mai rapidă dacă folosim `PreparedStatement`, deoarece nu mai trebuie să creăm câte un obiect de tip `Statement` pentru fiecare apel SQL, ci refolosim o singură instanţă precompilată furnizându-i doar alte argumente.

Crearea unui obiect de tip `PreparedStatement` se realizează prin intermediul metodei **prepareStatement** a clasei `Connection`, specificân ca argument o secvenţă SQL ce conţine căte un semn de întrebare pentru fiecare parametru de intrare:

```
Connection con = DriverManager.getConnection(url);
String sql = "UPDATE persoane SET nume=? WHERE cod=?";
Statement pstmt = con.prepareStatement(sql);
```

Obiectul va *pstmt* conţine o comandă SQL precompilată care este trimisă imediat către baza de date, unde va aştepta parametri de intrare pentru a putea fi executată.

Trimiterea parametrilor se realizează prin metode de tip **setXXX**, unde XXX este tipul corespunzător parametrului, iar argumentele metodei sunt *numărul de ordine* al parametrului de intrare (al semnului de întrebare) şi *valoarea* pe care dorim să o atribuim.

```
pstmt.setString(1, "Ionescu");
pstmt.setInt(2, 100);
```

După stabilirea parametrilor de intrare secvenţa SQL poate fi executată. Putem apoi stabili alte valori de intrare şi refolosi obiectul `PreparedStatement` pentru execuţii repetate ale comenzii SQL. Este însă posibil ca SGBD-ul folosit să nu suporte acest tip de operaţiune şi să nu reţină obiectul precompilat pentru execuţii ulterioare. In această situaţie folosirea interfeţei `PreparedStatement` în loc de `Statement` nu va îmbunătăţi în nici un fel performanţa codului, din punctul de vedere al vitezei de execuţie a acestuia.

Execuţia unei secvenţe SQL folosind un obiect `PreparedStatement` se realizează printr-una din metodele **executeQuery**, **executeUpdate** sau **execute**, semnificaţiile lor fiind aceleaşi ca şi în cazul obiectelor de tip `Statement`, cu singura deosebire că în cazul de faţă ele nu au nici un argument.

```
String sql = "UPDATE persoane SET nume=? WHERE cod=?";
Statement pstmt = con.prepareStatement(sql);
pstmt.setString(1, "Ionescu");
pstmt.setInt(2, 100);
pstmt.executeUpdate();

pstmt.setString(1, "Popescu");
pstmt.setInt(2, 200);
pstmt.executeUpdate();

sql = "SELECT * from persoane WHERE cod >= ?";
pstmt = con.prepareStatement(sql);
pstmt.setInt(1, 100);
ResultSet rs = pstmt.executeQuery();
```

Fiecărui tip Java îi corespunde un tip generic SQL. Este responsabilitatea programatorului să se asigure că foloseşte metoda adecvată de tip `setXXX` la stabilirea valorii unui parametru de intrare. Lista tuturor tipurilor generice disponibile, numite şi *tipuri JDBC*, este definită de clasa **Types**, prin constantelor declarate de aceasta. Metoda **setObject** permite specificarea unor valori pentru parametrii de intrare, atunci când dorim să folosim maparea implicită între tipurile Java şi cele JDBC sau atunci când dorim să precizăm explicit un tip JDBC.

```
pstmt.setObject(1, "Ionescu", Types.CHAR);
pstmt.setObject(2, 100, Types.INTEGER); // sau doar
pstmt.setObject(2, 100);
```

Folosind metoda **setNull** putem să atribuim unui parametru de intrare valoare SQL `NULL`, trebuind însă să specificăm şi tipul de date al coloanei în care vom scrie această valoare. Acelaşi lucru poate fi realizat cu metode de tipul `setXXX` dacă argumentul folosit are valoarea `null`.

```
pstmt.setNull(1, Types.CHAR);
pstmt.setInt(2, null);
```

Cu ajutorul metodelor **setBytes** sau **setString** avem posibilitatea de a specifica date de orice dimensiuni ca valori pentru anumite articole din baza de date. Există însă situaţii când este de preferat ca datele de mari dimensiuni să fie transferate pe "bucăţi" de o anumită dimensiune. Pentru a realiza

acest lucru API-ul JDBC pune la dispoziţie metodele **setBinaryStream**, **setAsciiStream** şi **setUnicodeStream** care ataşează un flux de intrare pe octeţi, caractere ASCII, respectiv UNICODE, unui parametru de intrare. Pe măsură ce sunt citite date de pe flux, ele vor fi atribuite parametrului. Exemplul de mai jos ilustrează acest lucru, atribuind coloanei *continut* conţinutul unui anumit fişier:

```
File file = new File("date.txt");
int fileLength = file.length();
InputStream fin = new FileInputStream(file);
java.sql.PreparedStatement pstmt = con.prepareStatement(
  "UPDATE fisiere SET continut = ? WHERE nume = 'date.txt'");
pstmt.setUnicodeStream (1, fin, fileLength);
pstmt.executeUpdate();
```

La execuţia secvenţei, fluxul de intrare va fi apelat repetat pentru a furniza datele ce vor fi scrise în coloana *continut* a articolului specificat. Observaţi că este necesar înă să ştim dinainte dimensiunea datelor ce vor fi scrise, acest lucru fiind solicitat de unele tipuri de baze de date.

15.3.3 Interfaţa CallableStatement

Interfaţa `CallableStatement` este derivată din `PreparedStatement`, instanţele de acest tip oferind o modalitate de a apela o procedură stocată într-o bază de date, într-o manieră standar pentru toate SGBD-urile.

Crearea unui obiect `CallableStatement` se realizează prin metoda **prepareCall** a clasei `Connection`:

```
Connection con = DriverManager.getConnection(url);
CallableStatement cstmt = con.prepareCall(
  "{call proceduraStocata(?, ?)}");
```

Trimiterea parametrilor de intrare se realizează întocmai ca la `PreparedStatement`, cu metode de tip **setXXX**. Dacă procedura are şi parametri de ieşire (valori returnate), aceştia vor trebui înregistraţi cu metoda **registerOutParameter** înainte de execuţia procedurii. Obţinerea valorilor rezultate în parametrii de ieşie se va face cu metode de tip **getXXX**.

```
CallableStatement cstmt = con.prepareCall(
```

```
  "{call calculMedie(?)}");
cstmt.registerOutParameter(1, java.sql.Types.FLOAT);
cstmt.executeQuery();
float medie = cstmt.getDouble(1);
```

Este posibil ca un parametru de intrare să fie şi parametru de ieşire. In acest caz el trebuie să primească o valoare cu `setXXX` şi, de asemenea, va fi înregistrat cu `registerOutParameter`, tipurile de date specificate trebuind să coincidă.

15.3.4 Obţinerea şi prelucrarea rezultatelor

15.3.5 Interfaţa ResultSet

In urma execuţie unei interogări SQL rezultatul va fi reprezentat printr-un obiect de tip **ResultSet**, ce va conţine toate liniile ce satisfac condiţiile impuse de comanda SQL. Forma generală a unui `ResultSet` este tabelară, având un număr de coloane şi de linii, funcţie de secvenţa executată. De asemenea, obiectul va conţine şi meta-datele interogării cum ar fi denumirele coloanelor selectate, numărul lor, etc.

```
Statement stmt = con.createStatement();
String sql = "SELECT cod, nume FROM persoane";
ResultSet rs = stmt.executeQuery(sql);
```

Rezultatul interogării de mai sus va fi obiectul rs cu următoarea structură:

cod	**nume**
100	Ionescu
200	Popescu

Pentru a extrage informaţiile din această structură va trebui să parcurgem tabelul linie cu linie şi din fiecare să extragem valorile de pe coloane. Pentru acest lucru vom folosi metode de tip **getXXX**, unde XXX este tipul de dată al unei coloane iar argumentul primit indică fie numărul de ordine din cadrul tabelului, fie numele acestuia. Coloanele sunt numerotate de la stânga la dreapta, începând cu 1. In general, folosirea indexului coloanei în loc de numele său va fi mai eficientă. De asemenea, pentru maximă portabilitate se recomandă citirea coloanelor în ordine de la stânga la dreapta şi fiecare citire să se facă o singură dată.

Un obiect `ResultSet` foloseşte un cursor pentru a parcurge articolele rezultate în urma unei interogări. Iniţial acest cursor este poziţionat înaintea primei linii, fiecare apel al metodei **next** determinând trecerea la următoarea linie. Deoarece `next` returnează `false` când nu mai sunt linii de adus, uzual va fi folosită o buclă `while-loop` petru a itera prin articolele tabelului:

```
String sql = "SELECT cod, nume FROM persoane";
ResultSet rs = stmt.executeQuery(sql);
while (rs.next()) {
  int cod = r.getInt("cod");
  String nume = r.getString("nume");
  /* echivalent:
  int cod = r.getInt(1);
  String nume = r.getString(2);
  */
  System.out.println(cod + ", " + nume);
}
```

Implicit, un tabel de tip `ResultSet` nu poate fi modificat iar cursorul asociat nu se deplasează decât înainte, linie cu linie. Aşadar, putem itera prin rezultatul unei interogări o singură dată şi numai de la prima la ultima linie. Este însă posibil să creăm `ResultSet`-uri care să permită modificarea sau deplasarea în ambele sensuri. Exemplul următor va folosi un cursor care este modificabil şi nu va reflecta schimbările produse de alţi utilizatori după crearea sa:

```
Statement stmt = con.createStatement(
        ResultSet.TYPE_SCROLL_INSENSITIVE,
        ResultSet.CONCUR_UPDATABLE);
String sql = "SELECT cod, nume FROM persoane";
ResultSet rs = stmt.executeQuery(sql);
```

Dacă un `ResultSet` foloseşte un cursor modificabil şi care poate naviga în ambele sensuri, atunci are la dispoziţie o serie de metode ce se bazează pe acest suport:

- **absolute** - Deplasează cursorul la o anumită linie specificată absolut;
- **updateXXX** - Actualizează valoarea unei coloane din linia curentă, unde XXX este un tip de date.

- **updateRow** - Transferă actualizările făcute liniei în baza de date.
- **moveToInsertRow** - deplasează cursorul la o linie spceială, numită *linie nouă*, utilizateă pentru a introduce noi articole în baza de date. Linia curentă anterioară a cursorului va fi memorată pentru a se putea reveni la ea.
- **insertRow** - inserează articolul din zona *linie nouă* în baza de date; cursorul trebuie să fie poziţionat le linia nouă la execuţia acestei operaţiuni.
- **moveToCurrentRow** - revine la linia curentă din tabel.
- **deleteRow** - şterge linia curentă din tabel şi din baza de date; nu poate fi apelată când cursorul este în modul *linie nouă*.

Nu toate sistemele de gestiune a bazelor de date oferă suport pentru folosirea cursoarelor care pot fi modificate. Pentru a determina dacă baza de date permite acest lucru pot fi utilizate metodele **supportsPositionedUpdate** şi **supportsPositionedDelete** ale clasei `DatabaseMetaData`. In cazul în care acest lucru este permis, este responsabilitatea driver-ului bazei de date să asigure rezolvarea problemelor legate de actualizarea concurentă a unui cursor, astfel încât să nu apară anomalii.

15.3.6 Exemplu simplu

In continuare vom da un exemplul simplu de utilizare a claselor de bază menţionate anterior. Programul va folosi o bază de date MySql, ce conţine un tabel numit *persoane*, având coloanele: *cod*, *nume* şi *salariu*. Scriptul SQL de creare a bazei este:

```
create table persoane(cod integer, nume char(50), salariu double);
```

Aplicaţia va goli tabelul cu persoane, după care va adăuga aleator un număr de articole, va efectua afişarea lor şi calculul mediei salariilor.

Listing 15.1: Exemplu simplu de utilzare JDBC

```
import java.sql.*;

public class TestJdbc {
  public static void main (String[] args) {
```

```
String url = "jdbc:mysql://localhost/test" ;
try{
  Class.forName("com.mysql.jdbc.Driver");
} catch(ClassNotFoundException e) {
  System.out.println("Eroare incarcare driver!\n" + e);
  return;
}
try{
  Connection con = DriverManager.getConnection(url);

  // Golim tabelul persoane
  String sql = "DELETE FROM persoane";
  Statement stmt = con.createStatement();
  stmt.executeUpdate(sql);

  // Adaugam un numar de persoane generate aleator
  // Tabelul persoane are coloanele (cod, nume, salariu)
  int n = 10;
  sql = "INSERT INTO persoane VALUES (?, ?, ?)";
  PreparedStatement pstmt = con.prepareStatement(sql);
  for(int i=0; i<n; i++) {
    int cod = i;
    String nume = "Persoana" + i;
    double salariu = 100 + Math.round(Math.random() *
       900);
    // salariul va fi intre 100 si 1000
    pstmt.setInt(1, cod);
    pstmt.setString(2, nume);
    pstmt.setDouble(3, salariu);
    pstmt.executeUpdate();
  }

  // Afisam persoanele ordonate dupa salariu
  sql = "SELECT * FROM persoane ORDER BY salariu";
  ResultSet rs = stmt.executeQuery(sql);
  while (rs.next())
    System.out.println(rs.getInt("cod") + ", " +
                       rs.getString("nume") + ", " +
                       rs.getDouble("salariu"));


  // Calculam salariul mediu
  sql = "SELECT avg(salariu) FROM persoane";
  rs = stmt.executeQuery(sql);
  rs.next();
```

```
      System.out.println("Media: " + rs.getDouble(1));

      // Inchidem conexiunea
      con.close();

    } catch(SQLException e) {
      e.printStackTrace();
    }
  }
}
```

15.4 Lucrul cu meta-date

15.4.1 Interfaţa DatabaseMetaData

După realizarea unui conexiuni la o bază de date, putem apela metoda **getMetaData** pentru a afla diverse informaţii legate de baza respectivă, aşa numitele *meta-date* (”date despre date”); Ca rezult al apelului metodei, vom obţine un obiect de tip **DatabaseMetaData** ce oferă metode pentru determinarea tabelelor, procedurilor stocate, capabilităţilor conexiunii, gramaticii SQL suportate, etc. ale bazei de date.

Programul următor afişează numele tuturor tabelelor dintr-o bază de dat înregistrată în ODBC.

Listing 15.2: Folosirea interfeţei DatabaseMetaData

```
import java.sql.*;

public class TestMetaData {
  public static void main (String[] args) {
    String url = "jdbc:odbc:test";
    try{
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    } catch(ClassNotFoundException e) {
      System.out.println("Eroare incarcare driver!\n" + e);
      return;
    }
    try{
      Connection con = DriverManager.getConnection(url);
      DatabaseMetaData dbmd = con.getMetaData();
      ResultSet rs = dbmd.getTables(null, null, null, null);
```

```
      while (rs.next())
        System.out.println(rs.getString("TABLE_NAME"));
      con.close();
    } catch(SQLException e) {
      e.printStackTrace();
    }
  }
}
```

15.4.2 Interfaţa ResultSetMetaData

Meta-datele unui `ResultSet` reprezintă informaţiile despre rezultatul conţinut în acel obiect cum ar fi numărul coloanelor, tipul şi denumirile lor, etc. Acestea sunt obţinute apelând metoda **getMetaData** pentru `ResultSet`-ul respectiv, care va returna un obiect de tip **ResultSetMetaData** ce poate fi apoi folosit pentru extragerea informaţiilor dorite.

```
ResultSet rs = stmt.executeQuery("SELECT * FROM tabel");
ResultSetMetaData rsmd = rs.getMetaData();
// Aflam numarul de coloane
int n = rsmd.getColumnCount();

// Aflam numele coloanelor
Sring nume[] = new String[n+1];
for(int i=1; i<=n; i++)
  nume[i] = rsmd.getColumnName(i);
```

Capitolul 16

Lucrul dinamic cu clase

16.1 Incărcarea claselor în memorie

După cum ştim execuţia unei aplicaţii Java este realizată de către maşina virtuală Java (JVM), aceasta fiind responsabilă cu interpretarea codului de octeţi rezultat în urma compilării. Spre deosebire de alte limbaje de programare cum ar fi C sau C++, un program Java compilat nu este descris de un fişier executabil ci de o mulţime de fişiere cu extensia `.class` corespunzătoare fiecărei clase a programului. In plus, aceste clase nu sunt înărcate toate în memorie la pornirea aplicaţiei, ci sunt înărcate pe parcursul execuţie acesteia atunci când este nevoie de ele, momentul efectiv în care se realizează acest lucru depinzând de implementarea maşinii virtuale.

Ciclul de viaţă al unei clase are aşadar următoarele etape:

1. *Incărcarea* - Este procesul regăsirii reprezentării binare a unei clase (fişierul .class) pe baza numelui complet al acestuia şi încărcarea acesteia în memorie. In urma acestui proces, va fi instanţiat un obiect de tip **java.lang.Class**, corespunzător clasei respective. Operaţiunea de încărcare a unei clase este realizată la un moment ce precede prima utilizare efectivă a sa.

2. *Editarea de legături* - Specifică incorporarea noului tip de date în JVM pentru a putea fi utlizat.

3. *Iniţializarea* - Constă în execuţia blocurilor statice de iniţializare şi iniţializarea variabilelor de clasă.

4. *Descărcarea* - Atunci când nu mai există nici o referinţă de tipul clasei respective, obiectul de tip `Class` creat va fi marcat pentru a fi eliminat din memorie de către *garbage collector*.

Incărcarea claselor unei aplicaţii Java în memorie este realizată prin intermediul unor obiecte pe care le vom numi generic *class loader*. Acestea sunt de două tipuri:

1. *Class loader-ul primordial (eng. bootstrap)* - Reprezintă o parte integrantă a maşinii virtuale, fiind responsabil cu încărcarea claselor standard din distribuţia Java.

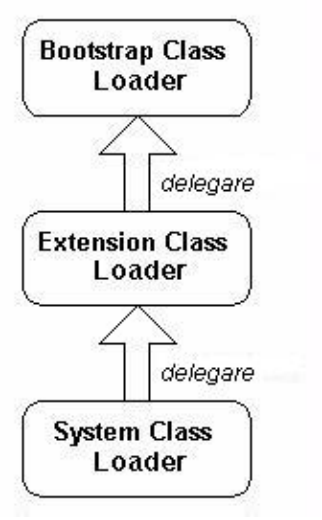
2. *Class loader-e proprii* - Acestea nu fac parte intrinsecă din JVM şi sunt instanţe ale clasei **java.lang.ClassLoader**. Aceasta este o clasă abstractă, tipul efectiv al obiectului fiind aşadar derivat din aceasta.

După cum vom vedea, la execuţia unui program Java vor fi create implicit două obiecte de tip `ClassLoader` pentru încărcarea în memorei a claselor proprii ale aplicaţiei. Există însă posibilitarea de a crea noi tipuri derivate din `ClassLoader` specializate pentru încărcarea claselor conform unor specificaţii anume care să realizeze diverse optimizări. Astfel, încărcarea unei clase poate determina încărcarea unor altor clase care sigur vor fi folosite împreună cu prima, sau a unor resurse ce sunt necesare funcţionării acesteia, etc.

Incepând cu versiunea 1.2 de Java, a fost introdus un model de tip *delegat*, în care class loader-ele sunt dispuse ierarhic într-un arbore, rădăcina acestuia fiind class loader-ul primordial. Fiecare instanţa de tip `ClassLoader` va avea aşadar un *părinte* (evident, mai puţin rădăcina), acesta fiind specificat la crearea sa. In momentul când este solicitată încărcarea unei clase, un class-loader poate delega în primul rând operaţiunea de încărcare părintelui său care va delega la rândul său cererea mai departe până la class loader-ul primordial sau până unul din aceştia reuşeşte să o încarce. Abia în cazul în care nici unul din aceştia nu a reuşit, va încerca să execute operaţiunea de încărcare a clasei. Dacă nici ea nu va reuşi, va fi aruncată o excepţie de tipul **ClassNotFoundException**. Deşi acest comportament nu este obligatoriu, în multe situaţii el este de preferat, pentru a minimiza încărcarea aceleiaşi clase de mai multe ori, folosind class loader-e diferite.

Implicit, Java 2 JVM oferă trei class loader-e, unul primordial şi două proprii, cunoscute sub numele de:

- *Boostrap Class Loader* - Class loader-ul primordial. Acesta este responsabil cu încărcarea claselor din distribuţia Java standard (cele din pachetele `java.*`, `javax.*`, etc.).
- *Extension Class Loader* - Utilizat pentru încărcarea claselor din directoarele extensiilor JRE.
- *System Class Loader* - Acesta este responsabil cu încărcarea claselor proprii aplicaţiilor Java (cele din CLASSPATH). Tipul acestuia este **java.lang.URLClassLoader**.



Intrucât tipurile de date Java pot fi încărcate folosind diverse instanţe de tip `ClassLoader`, fiecare obiect `Class` va reţine class loader-ul care a fost folosit pentru încărcare, acesta putând fi obţinut cu metoda **getClassLoader**.

Incărcarea *dinamică* a unei clase în memorie se referă la faptul că nu cunoaşţem tipul acesteia decât la execuţia preogramului, moment în care putem solicita încărcarea sa, specificând numele său complet prin intermediul unui şir de caractere. Acest lucru poate fi realizat prin mai multe modalităţi, cele mai comune metode fiind:

- **loadClass** apelată pentru un obiect de tip `ClassLoader`

```
ClassLoader loader = new MyClassLoader();
loader.loadClass("NumeCompletClasa");
```

- **Class.forName**
  Această metoda va încărca respectiva clasă folosind class loader-ul obiectului curent (care o apelează):

```
Class c = Class.forName("NumeCompletClasa");
// echivalent cu
ClassLoader loader = this.getClass().getClassLoader();
loader.loadClass("ClasaNecunoscuta");

// Clasele standard pot fi si ele incarcate astfel
Class  t = Class.forName("java.lang.Thread");
```

Dacă dorim să instanţiem un obiect dintr-o clasă încărcată dinamic putem folosi metoda **newInstance**, cu condiţia să existe constructorul fără argumente pentru clasa respectivă. După cum vom vedea în secţiunea următoare, mai există şi alte posibilităţi de a instanţia astfel de obiecte.

```
Class c = Class.forName("java.awt.Button");
Button b = (Button) c.newInstance();
```

Folosirea interfeţelor sau a claselor abstracte împreună cu încărcarea dinamică a claselor oferă un mecanism extrem de puternic de lucru în Java. Vom detalia acest lucru prin intermediul unui exepmplu. Să presupunem că dorim să creăm o aplicaţie care să genereze aleator un vector de numere după care să aplice o anumită funcţie acestui vector. Numele funcţiei care trebuie apelată va fi introdus de la tastatură, iar implementarea ei va fi conţinută într-o clasă a directorului curent. Toate funcţiile vor extinde clasa abstractă `Funcţie`. In felul acesta, aplicaţia poate fi extinsă cu noi funcţii fără a schimba codul ei, tot ce trebuie să facem fiind să scriem noi clase care extind `Functie` şi să implementăm metoda `executa`. Aceasta va returna 0 dacă metoda s-a executat corect, -1 în caz contrar.

Listing 16.1: Exemplu de încărcare dinamică a claselor

```
import java.util.*;
import java.io.*;

public class TestFunctii {

  public static void main(String args[]) throws IOException {
    // Generam un vector aleator
    int n = 10;
    int v[] = new int[n];
        Random rand = new Random();
    for(int i=0; i<n; i++)
      v[i] = rand.nextInt(100);

    // Citim numele unei functii
    BufferedReader stdin = new BufferedReader(
        new InputStreamReader(System.in));
    String numeFunctie = "";

    while (! numeFunctie.equals("gata")) {

      System.out.print("\nFunctie:");
          numeFunctie = stdin.readLine();

          try {
            // Incarcam clasa
            Class c = Class.forName(numeFunctie);

            // Cream un obiect de tip Functie
            Functie f = (Functie) c.newInstance();

            // Setam vectorul
            f.setVector(v); // sau f.v = v;

            //Executam functia
            int ret = f.executa();
            System.out.println("\nCod returnat: " + ret);

      } catch (ClassNotFoundException e) {
        System.err.println("Functie inexistenta !");
      } catch (InstantiationException e) {
        System.err.println("Functia nu poate fi instantiata !
            ");
      } catch (IllegalAccessException e) {
        System.err.println("Functia nu poate fi accesata !");
```

```
      }
    }
  }
}
```

Listing 16.2: Clasa abstractă ce descrie funcţia

```
public abstract class Functie {
  public int v[] = null;
  public void setVector(int[] v) {
    this.v = v;
  }
  public abstract int executa();
}
```

Listing 16.3: Un exemplu de funcţie

```
import java.util.*;
public class Sort extends Functie {
  public int executa() {
        if (v == null)
          return -1;
        Arrays.sort(v);
        for(int i=0; i<v.length; i++)
          System.out.print(v[i] + " ");
        return 0;
  }
}
```

Listing 16.4: Alt exemplu de funcţie

```
public class Max extends Functie {
  public int executa() {
        if (v == null)
          return -1;

        int max = v[0];
        for(int i=1; i<v.length; i++)
          if (max < v[i])
            max = v[i];
        System.out.print(max);
        return 0;
```

```
  }
}
```

Un obiect de tip `URLClassLoader` menţine o listă de adrese URL de unde va încerca să încarce în memorie clasa al cărei nume îl specificăm ca argument al metodelor de mai sus. Implicit, la crearea class loader-ului această listă este completată cu informaţiile din variabila sistem `CLASSPATH` sau cu cele specificate prin opţiunea `-classpath` la lansarea aplicaţiei. Folosind metoda **getURLs** putem afla aceste adrese, iar cu **addURL** putem adăuga o nouă adresă de căutare a claselor. Bineînţeles, adresele URL pot specifica şi directoare ale sistemului de fişiere local. Să presupunem că în directorul `c:\clase\demo` există clasa cu numele `Test`, aflată în pachetul `demo` şi dorim să o încărcăm dinamic în memorie:

```
  // Obtinem class loaderul curent
  URLClassLoader urlLoader =
    (URLClassLoader) this.getClass().getClassLoader();
  // Adaugam directorul sub forma unui URL
  urlLoader.addURL(new File("c:\\clase").toURL());
  // Incarcam clasa
  urlLoader.loadClass("demo.Test");
```

După ce o clasă a fost încărcată folosind un class loader, ea nu va mai putea fi descărcată explicit din memorie. In cazul în care dorim să avem posibilitatea de a o reîncărca, deoarece a fost modificată şi recompilată, trebuie să folosim class-loadere proprii şi să instanţiem noi obiecte de tip `ClassLoader`, ori de câte ori dorim să forţăm reîncărcarea claselor. Crearea unui class loader propriu se face uzual prin extinderea clasei `URLClassLoader`, o variantă simplistă fiind prezentată mai jos:

```
public class MyClassLoader extends URLClassLoader{
  public MyClassLoader(URL[] urls){
    super(urls);
  }
}
```

Incărcarea claselor folosind clasa nou creată se va face astfel:

```
// La initializare
URLClassLoader systemLoader =
  (URLClassLoader) this.getClass().getClassLoader();
URL[] urls = systemLoader.getURLs();

// Cream class loaderul propriu
MyClassLoader myLoader = new MyClassLoader(urls);
myLoader.loadClass("Clasa");
...
// Dorim sa reincarcam clasa
myLoader.loadClass("Clasa"); // nu functioneaza !

// Cream alt class loader
MyClassLoader myLoader = new MyClassLoader(urls);
myLoader.loadClass("Clasa"); // reincarca clasa
```

16.2 Mecanismul reflectării

Mecanismul prin care o clasă, interfaţă sau obiect "reflectă" la momentul execuţiei structura lor internă se numeşte *reflectare* (eng. reflection), acesta punând la dispoziţie metode pentru:

- Determinarea clasei unui obiect.
- Aflarea unor informaţii despre o clasă (modificatori, superclasa, constructori, metode).
- Instanţierea unor clase al căror nume este ştiut abia la execuţie.
- Setarea sau aflarea atributelor unui obiect, chiar dacă numele acestora este ştiut abia la execuţie.
- Invocarea metodelor unui obiect al căror nume este ştiut abia la execuţie.
- Crearea unor vectori a căror dimensiune şi tip nu este ştiut decât la execuţie.

Suportul pentru reflectare este inclus în distribuţia standard Java, fiind cunoscut sub numele de *Reflection API* şi conţine următoarele clase:

- `java.lang.Class`
- `java.lang.Object`
- Clasele din pachetul **java.lang.reflect** şi anume:
  - `Array`
  - `Constructor`
  - `Field`
  - `Method`
  - `Modifier`

16.2.1 Examinarea claselor şi interfeţelor

Examinarea claselor şi interfeţelor se realizează cu metode ale clasei `java.lang.Class`, un obiect de acest tip putând să reprezinte atât o clasă cât şi o interfaţă, diferenţierea acestora făcându-se prin intermediul metodei `isInterface`.

Reflection API pune la dispoziţie metode pentru obţinerea următoarelor informaţii:

Aflarea instanţei `Class` corespunzător unui anumit obiect sau tip de date:

```
Class c = obiect.getClass();
Class c = java.awt.Button.class;
Class c = Class.forName("NumeClasa");
```

Tipurile primitive sunt descrise şi ele de instanţe de tip `Class` având forma *TipPrimitiv*`.class`: `int.class, double.class`, etc., diferenţierea lor făcându-se cu ajutorul metodei `isPrimitive`.

Aflarea numelui unei clase - Se realizează cu metoda `getName`:

```
Class clasa = obiect.getClass();
String nume = clasa.getName();
```

Aflarea modificatorilor unei clase - Se realizează cu metoda `getModifiers`, aceasta returnând un număr întreg ce codifică toţi modificatorii clasei. Pentru a determina uşor prezenţa unui anumit modificator se folosesc metodele statice ale clasei `Modifier isPublic, isAbstract` şi `isFinal`:

```
Class clasa = obiect.getClass();
int m = clasa.getModifiers();
String modif = "";
if (Modifier.isPublic(m))
  modif += "public ";
if (Modifier.isAbstract(m))
  modif += "abstract ";
if (Modifier.isFinal(m))
  modif += "final ";
System.out.println(modif + "class" + c.getName());
```

Aflarea superclasei - Se realizează cu metoda `getSuperclass` ce returnează o instanţă de tip `Class`, corespunzătoare tipului de date al superclasei sau `null` pentru clasa `Object`.

```
Class c = java.awt.Frame.class;
Class s = c.getSuperclass();
System.out.println(s); // java.awt.Window

Class c = java.awt.Object.class;
Class s = c.getSuperclass(); // null
```

Aflarea interfeţelor implementate de o clasă sau extinse de o interfaţă - Se realizează cu metoda `getInterfaces`, ce returnează un vector de tip `Class[]`.

```
public void interfete(Class c) {
  Class[] interf = c.getInterfaces();
  for (int i = 0; i < interf.length; i++) {
    String nume = interf[i].getName();
    System.out.print(nume + " ");
```

```
    }
  }
  ...
  interfete(java.util.HashSet.class);
  // Va afisa interfetele implementate de HashSet:
  // Cloneable, Collection, Serializable, Set


  interfete(java.util.Set);
  // Va afisa interfetele extinse de Set:
  // Collection
```

Aflarea variabilelor membre - Se realizează cu una din metodele `getFields` sau `getDeclaredFields`, ce returnează un vector de tip `Field[]`, diferenţa între cele două constând în faptul că prima returnează *toate* variabilele membre, inclusiv cele moştenite, în timp ce a doua le returnează doar pe cele declarate în cadrul clasei. La rândul ei, clasa `Field` pune la dispoziţie metodele `getName`, `getType` şi `getModifiers` pentru a obţine numele, tipul, respectiv modificatorii unei variabile membru.

Cu ajutorul metodei `getField` este posibilă obţinerea unei referinţe la o variabilă mebră cu un anumit nume specificat.

Aflarea constructorilor - Se realizează cu metodele `getConstructors` sau `getDeclaredConstructors`, ce returnează un vector de tip `Constructor[]`. Clasa `Constructor` pune la dispoziţie metodele `getName`, `getParameterTypes`, `getModifiers`, `getExceptionTypes` pentru a obţine toate informaţiile legate de respectivul constructor.

Cu ajutorul metodei `getConstructor` este posibilă obţinerea unei referinţe la constructor cu o signatură specificată.

Aflarea metodelor - Se realizează cu metodele `getMethods` sau `getDeclaredMethods`, ce returnează un vector de tip `Method[]`. Clasa `Method` pune la dispoziţie metodele `getName`, `getParameterTypes`, `getModifiers`, `getExceptionTypes`, `getReturnType` pentru a obţine toate informaţiile legate

de respectiva metodă.

Cu ajutorul metodei `getMethod` este posibilă obţinerea unei referinţe la o metodă cu o signatură specificată.

Aflarea claselor imbricate - Se realizează cu metodele `getClasses` sau `getDeclaredClasses`, ce returnează un vector de tip `Class[]`.

Aflarea clasei de acoperire - Se realizează cu metoda `getDeclaringClass`. Această metodă o regăsim şi în clasele `Field`, `Constructor`, `Method`, pentru acestea returnând clasa cărei îi aparţine variabila, constructorul sau metoda respectivă.

16.2.2 Manipularea obiectelor

Pe lângă posibilitatea de a examina structura unei anumite clase la momentul execuţiei, folosind *Reflection API* avem posibilitatea de a lucra dinamic cu obiecte, bazându-ne pe informaţii pe care le obţinem abia la execuţie.

Crearea obiectelor

După cum ştim, crearea obiectelor se realizează cu operatorul `new` urmat de un apel la un constructor al clasei pe care o instanţiem. In cazul în care numele clasei nu este cunoscut decât la momentul execuţiei nu mai putem folosi această metodă de instanţiere. In schimb, avem la dispoziţie alte două variante:

- Metoda **newInstance** din clasa **java.lang.Class**
  Aceasta permite instanţierea unui obiect folosind constructorul fără argumente al acestuia. Dacă nu există un astfel de constructor sau nu este accesibil vor fi generate excepţii de tipul `InstantiationException`, respectiv `IllegalAccessException`.

```
Class c = Class.forName("NumeClasa");
Object o = c.newInstance();

// Daca stim tipul obiectului
```

```
Class c = java.awt.Point.class;
Point p = (Point) c.newInstance();
```

- Metoda **newInstance** din clasa **Constructor**
  Aceasta permite instanţierea unui obiect folosind un anumit constructor, cel pentru care se face apelul. Evident, această soluţie presupune în primul rând obţinerea unui obiect de tip `Constructor` cu o anumită signatură, apoi specificarea argumentelor la apelarea sa. Să rescriem exemplul de mai sus, apelând constructorul cu două argumente al clasei `Point`.

```
Class clasa = java.awt.Point.class;

// Obtinem constructorul dorit
Class[] signatura = new Class[] {int.class, int.class};
Constructor ctor = clasa.getConstructor(signatura);

// Pregatim argumentele
// Ele trebuie sa fie de tipul referinta corespunzator
Integer x = new Integer(10);
Integer y = new Integer(20);
Object[] arg = new Object[] {x, y};

// Instantiem
Point p = (Point) ctor.newInstance(arg);
```

Excepţii generate de metoda `newInstance` sunt: `InstantiationException`, `IllegalAccessException`, `IllegalArgumentException` şi `InvocationTargetException`. Metoda `getConstructor` poate provoca excepţii de tipul `NoSuchMethodException`.

Invocarea metodelor

Invocarea unei metode al cărei nume îl cunoaştem abia la momentul execuţiei se realizează cu metoda **invoke** a clasei **Method**. Ca şi în cazul constructorilor, trebuie să obţinem întâi o referinţă la metoda cu signatura corespunzătoare şi apoi să specificăm argumentele. In plus, mai putem obţine valoarea returnată. Să presupunem că dorim să apelăm metoda `contains`

a clasei `Rectangle` care determină dacă un anumit punct se găseşte în interiorul drepunghiului. Metoda `contains` are mai multe variante, noi o vom apela pe cea care acceptă un argument de tip `Point`.

```
Class clasa = java.awt.Rectangle.class;
Rectangle obiect = new Rectangle(0, 0, 100, 100);

// Obtinem metoda dorita
Class[] signatura = new Class[] {Point.class};
Method metoda = clasa.getMethod("contains", signatura);

// Pregatim argumentele
Point p = new Point(10, 20);
Object[] arg = new Object[] {p};

// Apelam metoda
metoda.invoke(obiect, arg);
```

Dacă numărul argumentelor metodei este 0, atunci putem folosi valoarea `null` în locul vectorilor ce reprezintă signatura, respectiv parametri de apelare ai metodei.

Excepţiile generate de metoda `invoke` sunt: `IllegalAccessException` şi `InvocationTargetException`. Metoda `getMethod` poate provoca excepţii de tipul `NoSuchMethodException`.

Setarea şi aflarea variabilelor membre

Pentur setarea şi aflarea valorilor variabilelor membre sunt folosite metodele **set** şi **get** ale clasei **Field**. Să presupunem că dorim să setăm variabila `x` a unui obiect de tip `Point` şi să obţinem valoarea variabilei `y` a aceluiaşi obiect:

```
Class clasa = java.awt.Point.class;
Point obiect = new Point(0, 20);

// Obtinem variabilele membre
Field x, y;
x = clasa.getField("x");
y = clasa.getField("y");
```

```
// Setam valoarea lui x
x.set(obiect, new Integer(10));

// Obtinem valoarea lui y
Integer val = y.get(obiect);
```

Excepţiile generate de metodele `get, set` sunt: `IllegalAccessException` şi `IllegalArgumentException`. Metoda `getField` poate provoca excepţii de tipul `NoSuchFieldException`.

Revenind la exemplul din secţiunea anterioară cu apelarea dinamică a unor funcţii pentru un vector, să presupunem că există deja un număr însemnat de clase care descriu diferite funcţii dar acestea nu extind clasa abstractă `Functie`. Din acest motiv, soluţia anterioară nu mai este viabilă şi trebuie să folosim apelarea metodei `executa` într-un mod dinamic.

Listing 16.5: Lucru dinamic cu metode şi variabile

```
import java.lang.reflect.*;
import java.util.*;
import java.io.*;

public class TestFunctii2 {

  public static void main(String args[]) throws IOException {
    // Generam un vector aleator
    int n = 10;
    int v[] = new int[n];
        Random rand = new Random();
    for(int i=0; i<n; i++)
      v[i] = rand.nextInt(100);

    // Citim numele unei functii
    BufferedReader stdin = new BufferedReader(
        new InputStreamReader(System.in));
    String numeFunctie = "";

    while (! numeFunctie.equals("gata")) {

      System.out.print("\nFunctie:");
          numeFunctie = stdin.readLine();
```

```
          try {
            // Incarcam clasa
            Class c = Class.forName(numeFunctie);

            // Cream un obiect de tip Functie
            Object f = c.newInstance();

            // Setam vectorul (setam direct variabila v)
            Field vector = c.getField("v");
            vector.set(f, v);

            // Apelam metoda 'executa'
            // Folosim null pentru ca nu avem argumente
            Method m = c.getMethod("executa", null);
            Integer ret = (Integer) m.invoke(f, null);
            System.out.println("\nCod returnat: " + ret);

      } catch (Exception e) {
          System.err.println("Eroare la apelarea functiei !");
      }
    }
  }
}
```

16.2.3 Lucrul dinamic cu vectori

Vectorii sunt reprezentaţi ca tip de date tot prin intermediul clasei `java.lang.Class`, diferenţierea făcându-se prin intermediul metodei `isArray`.

Tipul de date al elementelor din care este format vectorul va fi obţinut cu ajutorul metodei `getComponentType`, ce întoarce o referinţă de tip `Class`.

```
Point []vector = new Point[10];
Class c = vector.getClass();
System.out.println(c.getComponentType());
// Va afisa: class java.awt.Point
```

Lucrul dinamic cu obiecte ce reprezintă vectori se realizează prin intermediul clasei **Array**. Aceasta conţine o serie de metode statice ce permit:

- Crearea de noi vectori: `newInstance`
- Aflarea numărului de elemente: `getLength`

- Setarea / aflarea elementelor: `set`, `get`

Exemplul de mai jos creează un vector ce conţine numerele întregi de la 0 la 9:

```
Object a = Array.newInstance(int.class, 10);
for (int i=0; i < Array.getLength(a); i++)
  Array.set(a, i, new Integer(i));

for (int i=0; i < Array.getLength(a); i++)
  System.out.print(Array.get(a, i) + " ");
```