Fundamentele limbajului 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 domeniul 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 cunoscut 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 instalat 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 întotdeauna 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`: termină 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 în exemplul 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, să-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 făcută explicit, alocarea memoriei făcându-se automat în funcţie de numărul 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 procesul de creare a unei aplicaţii, 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 *Quick-Sort* 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 său pe parcursul execuţiei programului) atunci el va fi declarat de tipul `String`, altfel va fi declarat de tip `StringBuilder` sau `StringBuffer`, acestea din urmă punând 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` faţă 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 comandă 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 sa 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 al clasei în care se găseşte metoda `main`.

Să considerăm 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 număr 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 număr 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 găseşte 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").

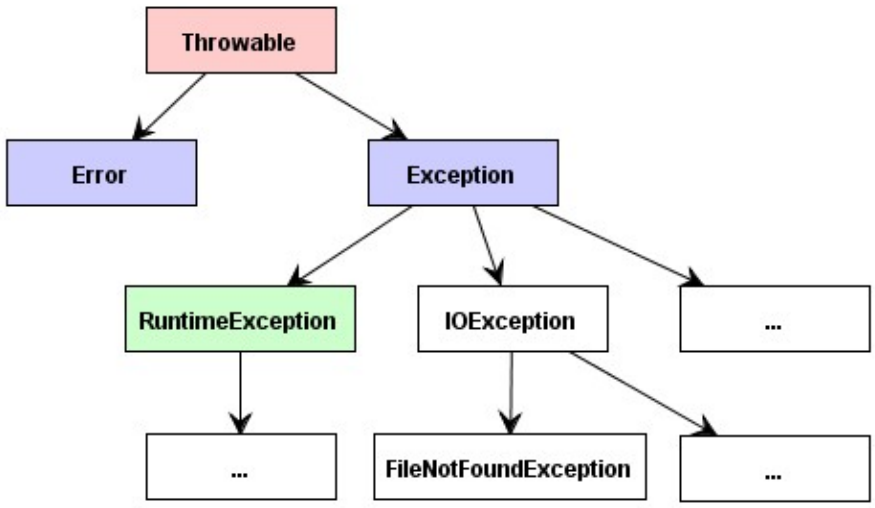
Pronderea unei excepţii se poate face fie la nivelul clasei specifice pentru acea excepţie, fie la nivelul uneia din superclasele sale, în funcţie de necesităţile programului, însă, cu cât clasa folosită este mai generică cu atât tratarea excepţiilor programul îşi pierde din flexibilitate.

```
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:

`Writer`	`OutputStream`
`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<100; 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 %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

Interfaţa grafică cu utilizatorul

6.1 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.

6.2 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 6.1: 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").

6.2.1 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`

6.2.2 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);
```

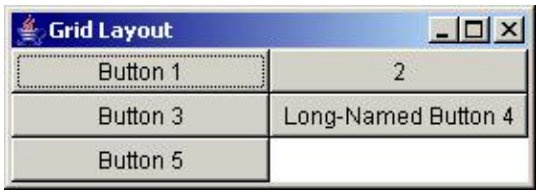
6.3 Gestionarea poziţionării

Să considerăm mai întâi un exemplu de program Java care afişează 5 butoane pe o fereastră:

Listing 6.2: 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`.

6.3.1 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.

6.3.2 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.

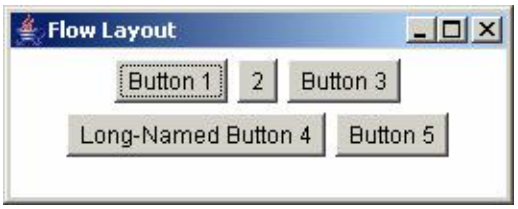
Listing 6.3: Gestionarul `FlowLayout`

```
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:



6.3.3 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 6.4: 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.

6.3.4 Gestionarul GridLayout

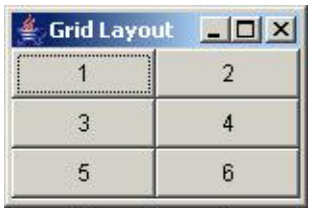
Gestionarul `GridLayout` organizează containerul ca un tabel cu rânduri şi coloane, componentele fiind plasate în celulele tabelului de la stânga la dreapta, începând cu primul rând. Celulele tabelului au dimensiuni egale iar o componentă poate ocupa doar o singură celulă. Numărul de linii şi coloane 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 6.5: 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.

6.3.5 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 6.6: 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

6.3.6 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 6.7: 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();
  }
}
```

6.3.7 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 6.8: Gruparea componentelor

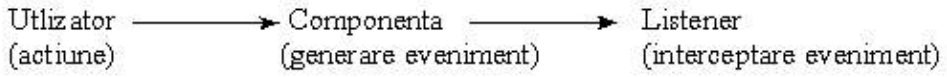
```
import java.awt.*;
public class TestPanel {
  public static void main(String args[]) {
    Frame f = new Frame("Test Panel");

    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();
  }
}
```

6.4 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.

6.4.1 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 6.9: 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 6.10: 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.

6.4.2 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.

6.4.3 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 6.11: 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 6.12: 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 6.13: 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();
  }
}
```

6.5 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*.

6.5.1 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).

6.5.2 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 6.14: 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;

6.5.3 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 6.15: 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();
  }
}
```

6.5.4 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 6.16: 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.

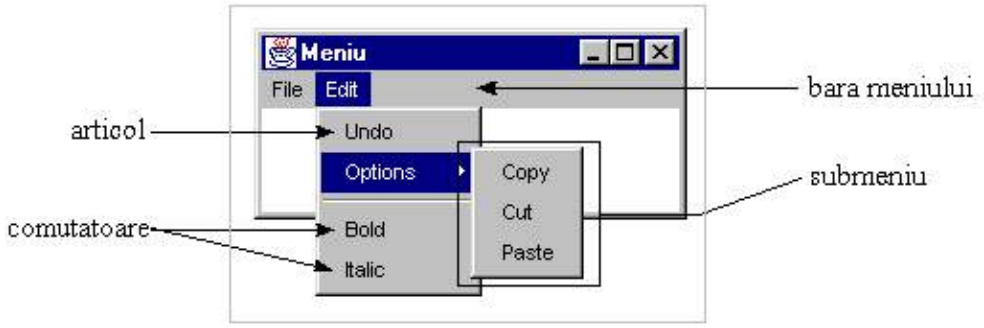
6.6 Folosirea meniurilor

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:



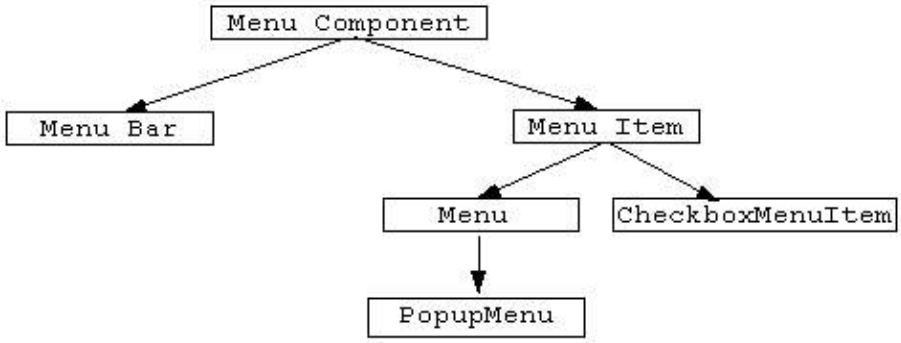
Exemplul de mai sus conţine o bară de meniuri formată din două meniuri principale *File* şi *Edit*. Meniul *Edit* conţine la rândul lui alt meniu (submeniu) *Options*, articolul *Undo* şi două comutatoare *Bold* şi *Italic*. Prin abuz de limbaj, vom referi uneori bara de meniuri a unei ferestre ca fiind 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`.

6.6.1 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 6.17: 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();
  }
}
```

6.6.2 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 6.18: 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();
  }
}
```

6.6.3 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 6.19: 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();
  }
}
```

6.6.4 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);
```

Capitolul 7

Appleturi

7.1 Introducere

Definiţie

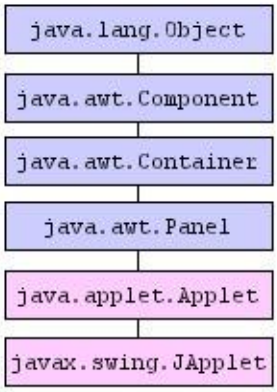
Un *applet* reprezintă un program Java de dimensiuni reduse ce gestionează o suprafaţă de afişare (container) care poate fi inclusă într-o pagină Web. Un astfel de program se mai numeşte *miniaplicatie*.

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`.

7.2 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`.

7.3 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 oprească 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() {
  }

}
```

7.4 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.

7.5 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 7.1: 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;
  }

}
```

7.6 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.

7.7 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 7.2: 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 7.3: 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 7.4: 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 7.5: 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.

7.8 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 7.6: 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 7.7: 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.

7.9 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 />
```

7.10 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.

7.11 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 7.8: 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();
  }
}
```