

Curs practic de Java

Cristian Frăsinaru

Cuprins

1	Introducere în Java	11
1.1	Ce este Java ?	11
1.1.1	Limbaajul de programare Java	11
1.1.2	Platforme de lucru Java	12
1.1.3	Java: un limbaaj compilat și interpretat	13
1.2	Primul program	14
1.3	Structura lexicală a limbaajului Java	16
1.3.1	Setul de caractere	16
1.3.2	Cuvinte cheie	16
1.3.3	Identificatori	17
1.3.4	Literali	17
1.3.5	Separatori	19
1.3.6	Operatori	19
1.3.7	Comentarii	20
1.4	Tipuri de date și variabile	21
1.4.1	Tipuri de date	21
1.4.2	Variabile	22
1.5	Controlul execuției	24
1.5.1	Instrucțiuni de decizie	24
1.5.2	Instrucțiuni de salt	25
1.5.3	Instrucțiuni pentru tratarea excepțiilor	26
1.5.4	Alte instrucțiuni	26
1.6	Vectori	26
1.6.1	Crearea unui vector	26
1.6.2	Tablouri multidimensionale	28
1.6.3	Dimensiunea unui vector	28
1.6.4	Copierea vectorilor	29

1.6.5	Sortarea vectorilor - clasa <code>Arrays</code>	29
1.6.6	Vectori cu dimensiune variabilă și eterogeni	30
1.7	Șiruri de caractere	30
1.8	Folosirea argumentelor de la linia de comandă	31
1.8.1	Transmiterea argumentelor	31
1.8.2	Primirea argumentelor	32
1.8.3	Argumente numerice	34
2	Obiecte și clase	35
2.1	Ciclul de viață al unui obiect	35
2.1.1	Crearea obiectelor	35
2.1.2	Folosirea obiectelor	37
2.1.3	Distrușgerea obiectelor	38
2.2	Crearea claselor	39
2.2.1	Declararea claselor	39
2.2.2	Extinderea claselor	40
2.2.3	Corpul unei clase	41
2.2.4	Constructorii unei clase	42
2.2.5	Declararea variabilelor	46
2.2.6	<code>this</code> și <code>super</code>	49
2.3	Implementarea metodelor	50
2.3.1	Declararea metodelor	50
2.3.2	Tipul returnat de o metodă	52
2.3.3	Trimiterea parametrilor către o metodă	53
2.3.4	Metode cu număr variabil de argumente	56
2.3.5	Supraîncărcarea și supradefinirea metodelor	57
2.4	Modificatori de acces	58
2.5	Membri de instanță și membri de clasă	59
2.5.1	Variabile de instanță și de clasă	59
2.5.2	Metode de instanță și de clasă	61
2.5.3	Utilitatea membrilor de clasă	62
2.5.4	Blocuri statice de inițializare	63
2.6	Clase imbricate	64
2.6.1	Definirea claselor imbricate	64
2.6.2	Clase interne	66
2.6.3	Identificare claselor imbricate	66
2.6.4	Clase anonime	67
2.7	Clase și metode abstracte	67

2.7.1	Declararea unei clase abstracte	68
2.7.2	Metode abstracte	68
2.8	Clasa Object	71
2.8.1	Orice clasă are o superclasă	71
2.8.2	Clasa Object	71
2.9	Conversii automate între tipuri	74
2.10	Tipul de date enumerare	75
3	Excepții	77
3.1	Ce sunt excepțiile ?	77
3.2	”Prinderea” și tratarea excepțiilor	78
3.3	”Aruncarea” excepțiilor	82
3.4	Avantajele tratării excepțiilor	85
3.4.1	Separarea codului pentru tratarea erorilor	85
3.4.2	Propagarea erorilor	87
3.4.3	Gruparea erorilor după tipul lor	89
3.5	Ierarhia claselor ce descriu excepții	90
3.6	Excepții la execuție	91
3.7	Crearea propriilor excepții	92
4	Intrări și ieșiri	95
4.1	Introducere	95
4.1.1	Ce sunt fluxurile?	95
4.1.2	Clasificarea fluxurilor	96
4.1.3	Ierarhia claselor pentru lucrul cu fluxuri	97
4.1.4	Metode comune fluxurilor	98
4.2	Folosirea fluxurilor	99
4.2.1	Fluxuri primitive	99
4.2.2	Fluxuri de procesare	100
4.2.3	Crearea unui flux	101
4.2.4	Fluxuri pentru lucrul cu fișiere	103
4.2.5	Citirea și scrierea cu buffer	105
4.2.6	Concatenarea fluxurilor	107
4.2.7	Fluxuri pentru filtrarea datelor	108
4.2.8	Clasele <code>DataInputStream</code> și <code>DataOutputStream</code>	109
4.3	Intrări și ieșiri formatate	110
4.3.1	Intrări formatate	110
4.3.2	Ieșiri formatate	111

4.4	Fluxuri standard de intrare și ieșire	111
4.4.1	Afisarea informațiilor pe ecran	112
4.4.2	Citirea datelor de la tastatură	112
4.4.3	Redirectarea fluxurilor standard	113
4.4.4	Analiza lexicală pe fluxuri (clasa <code>StreamTokenizer</code>)	115
4.5	Clasa <code>RandomAccessFile</code> (fișiere cu acces direct)	117
4.6	Clasa <code>File</code>	119
5	Interfețe	121
5.1	Introducere	121
5.1.1	Ce este o interfață ?	121
5.2	Folosirea interfețelor	122
5.2.1	Definirea unei interfețe	122
5.2.2	Implementarea unei interfețe	123
5.2.3	Exemplu: implementarea unei stive	124
5.3	Interfețe și clase abstracte	129
5.4	Moștenire multiplă prin interfețe	130
5.5	Utilitatea interfețelor	132
5.5.1	Crearea grupurilor de constante	132
5.5.2	Transmiterea metodelor ca parametri	133
5.6	Interfața <code>FilenameFilter</code>	134
5.6.1	Folosirea claselor anonime	137
5.7	Compararea obiectelor	138
5.7.1	Interfața <code>Comparable</code>	139
5.7.2	Interfața <code>Comparator</code>	141
5.8	Adaptori	142
6	Organizarea claselor	145
6.1	Pachete	145
6.1.1	Pachetele standard (J2SDK)	145
6.1.2	Folosirea membrilor unui pachet	146
6.1.3	Importul unei clase sau interfețe	147
6.1.4	Importul la cerere dintr-un pachet	148
6.1.5	Importul static	149
6.1.6	Crearea unui pachet	150
6.1.7	Denumirea unui pachet	151
6.2	Organizarea fișierelor	152
6.2.1	Organizarea fișierelor sursă	152

6.2.2	Organizarea unităților de compilare (.class)	154
6.2.3	Necesitatea organizării fișierelor	155
6.2.4	Setarea căii de căutare (CLASSPATH)	156
6.3	Arhive JAR	157
6.3.1	Folosirea utilitarului jar	158
6.3.2	Executarea aplicațiilor arhivate	159
7	Colecții	161
7.1	Introducere	161
7.2	Interfețe ce descriu colecții	162
7.3	Implementări ale colecțiilor	166
7.4	Folosirea eficientă a colecțiilor	168
7.5	Algoritmi polimorfici	170
7.6	Tipuri generice	171
7.7	Iteratori și enumerări	172
8	Serializarea obiectelor	177
8.1	Folosirea serializării	177
8.1.1	Serializarea tipurilor primitive	179
8.1.2	Serializarea obiectelor	180
8.1.3	Clasa ObjectOutputStream	180
8.1.4	Clasa ObjectInputStream	181
8.2	Obiecte serializabile	183
8.2.1	Implementarea interfeței Serializable	183
8.2.2	Controlul serializării	184
8.3	Personalizarea serializării obiectelor	187
8.3.1	Controlul versiunilor claselor	188
8.3.2	Securizarea datelor	193
8.3.3	Implementarea interfeței Externalizable	194
8.4	Clonarea obiectelor	196
9	Interfața grafică cu utilizatorul	199
9.1	Introducere	199
9.2	Modelul AWT	200
9.2.1	Componentele AWT	202
9.2.2	Suprafețe de afișare (Clasa Container)	204
9.3	Gestionarea poziționării	206
9.3.1	Folosirea gestionarilor de poziționare	207

9.3.2	Gestionarul FlowLayout	209
9.3.3	Gestionarul BorderLayout	210
9.3.4	Gestionarul GridLayout	211
9.3.5	Gestionarul CardLayout	212
9.3.6	Gestionarul GridBagLayout	214
9.3.7	Gruparea componentelor (Clasa Panel)	218
9.4	Tratarea evenimentelor	219
9.4.1	Exemplu de tratare a evenimentelor	221
9.4.2	Tipuri de evenimente	224
9.4.3	Folosirea adaptorilor și a claselor anonime	227
9.5	Folosirea ferestrelor	232
9.5.1	Clasa Window	232
9.5.2	Clasa Frame	233
9.5.3	Clasa Dialog	236
9.5.4	Clasa FileDialog	239
9.6	Folosirea meniurilor	242
9.6.1	Ierarhia claselor ce descriu meniuri	243
9.6.2	Tratarea evenimentelor generate de meniuri	246
9.6.3	Meniuri de context (popup)	247
9.6.4	Acceleratori (Clasa MenuShortcut)	250
9.7	Folosirea componentelor AWT	250
9.7.1	Clasa Label	251
9.7.2	Clasa Button	252
9.7.3	Clasa Checkbox	253
9.7.4	Clasa CheckboxGroup	255
9.7.5	Clasa Choice	257
9.7.6	Clasa List	259
9.7.7	Clasa ScrollBar	261
9.7.8	Clasa ScrollPane	262
9.7.9	Clasa TextField	263
9.7.10	Clasa TextArea	265
10	Desenarea	269
10.1	Conceptul de desenare	269
10.1.1	Metoda paint	270
10.1.2	Suprafețe de desenare - clasa Canvas	271
10.2	Contextul grafic de desenare	274
10.2.1	Proprietățile contextului grafic	275

10.2.2	Primitive grafice	275
10.3	Folosirea fonturilor	276
10.3.1	Clasa Font	277
10.3.2	Clasa FontMetrics	279
10.4	Folosirea culorilor	282
10.5	Folosirea imaginilor	286
10.5.1	Afișarea imaginilor	287
10.5.2	Monitorizarea încărcării imaginilor	289
10.5.3	Mecanismul de ”double-buffering”	291
10.5.4	Salvarea desenelor în format JPEG	291
10.5.5	Crearea imaginilor în memorie	292
10.6	Tipărirea	293
11	Swing	299
11.1	Introducere	299
11.1.1	JFC	299
11.1.2	Swing API	300
11.1.3	Asemănări și deosebiri cu AWT	301
11.2	Folosirea ferestrelor	304
11.2.1	Ferestre interne	305
11.3	Clasa JComponent	307
11.4	Arhitectura modelului Swing	310
11.5	Folosirea modelelor	310
11.5.1	Tratarea evenimentelor	314
11.6	Folosirea componentelor	316
11.6.1	Componente atomice	316
11.6.2	Componente pentru editare de text	316
11.6.3	Componente pentru selectarea unor elemente	319
11.6.4	Tabele	324
11.6.5	Arbori	329
11.6.6	Containere	332
11.6.7	Dialoguri	335
11.7	Desenarea	336
11.7.1	Metode specifice	336
11.7.2	Considerații generale	338
11.8	Look and Feel	340

12 Fire de execuție	343
12.1 Introducere	343
12.2 Crearea unui fir de execuție	344
12.2.1 Extinderea clasei Thread	345
12.2.2 Implementarea interfeței Runnable	347
12.3 Ciclul de viață al unui fir de execuție	352
12.3.1 Terminarea unui fir de execuție	355
12.3.2 Fire de execuție de tip "daemon"	357
12.3.3 Stabilirea priorităților de execuție	358
12.3.4 Sincronizarea firelor de execuție	362
12.3.5 Scenariul producător / consumator	362
12.3.6 Monitoare	367
12.3.7 Semafoare	369
12.3.8 Probleme legate de sincronizare	371
12.4 Gruparea firelor de execuție	373
12.5 Comunicarea prin fluxuri de tip "pipe"	376
12.6 Clasele Timer și TimerTask	378
13 Programare în rețea	383
13.1 Introducere	383
13.2 Lucrul cu URL-uri	385
13.3 Socket-uri	387
13.4 Comunicarea prin conexiuni	388
13.5 Comunicarea prin datagrame	393
13.6 Trimiterea de mesaje către mai mulți clienți	397
14 Appleturi	401
14.1 Introducere	401
14.2 Crearea unui applet simplu	402
14.3 Ciclul de viață al unui applet	404
14.4 Interfața grafică cu utilizatorul	406
14.5 Definirea și folosirea parametrilor	408
14.6 Tag-ul APPLET	410
14.7 Folosirea firelor de execuție în appleturi	412
14.8 Alte metode oferite de clasa Applet	416
14.9 Arhivarea appleturilor	420
14.10 Restricții de securitate	421
14.11 Appleturi care sunt și aplicații	421

15	Lucrul cu baze de date	423
15.1	Introducere	423
15.1.1	Generalități despre baze de date	423
15.1.2	JDBC	424
15.2	Conectarea la o bază de date	425
15.2.1	Inregistrarea unui driver	426
15.2.2	Specificarea unei baze de date	427
15.2.3	Tipuri de drivere	428
15.2.4	Realizarea unei conexiuni	430
15.3	Efectuarea de secvențe SQL	431
15.3.1	Interfața Statement	432
15.3.2	Interfața PreparedStatement	434
15.3.3	Interfața CallableStatement	437
15.3.4	Obținerea și prelucrarea rezultatelor	438
15.3.5	Interfața ResultSet	438
15.3.6	Exemplu simplu	440
15.4	Lucrul cu meta-date	442
15.4.1	Interfața DatabaseMetaData	442
15.4.2	Interfața ResultSetMetaData	443
16	Lucrul dinamic cu clase	445
16.1	Încărcarea claselor în memorie	445
16.2	Mecanismul reflectării	452
16.2.1	Examinarea claselor și interfețelor	453
16.2.2	Manipularea obiectelor	456
16.2.3	Lucrul dinamic cu vectori	460

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 comunități a dezvoltatorilor de software, impunându-se prin calități deosebite cum ar fi simplitate, robustețe și nu în ultimul rând portabilitate. Denumită inițial *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 aplicații pentru toate segmentele industriei software.

1.1.1 Limbajul de programare Java

Înainte 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 populare opțiuni pentru dezvoltarea de aplicații, indiferent de domeniu sau de complexitatea lor.

- **Simplitate** - elimină supraîncărcarea operatorilor, moștenirea multiplă și toate "facilitățile" ce pot provoca scrierea unui cod confuz.
- **Ușurință** în crearea de aplicații complexe ce folosesc programarea în rețea, fire de execuție, interfață grafică, baze de date, etc.
- **Robustețe** - elimină sursele frecvente de erori ce apar în programare prin renunțarea la pointeri, administrarea automată a memoriei și 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 secvențelor periculoase, impunerea unor reguli stricte pentru rularea proceselor la distanță, etc.
- **Neutralitate arhitecturală** - comportamentul unei aplicații Java nu depinde de arhitectura fizică a mașinii pe care rulează.
- **Portabilitate** - Java este un limbaj independent de platforma de lucru, aceeași aplicație rulând fără nici o modificare și fără a necesita recompilarea ei pe sisteme de operare diferite cum ar fi Windows, Linux, Mac OS, Solaris, etc. lucru care aduce economii substanțiale firmelor dezvoltatoare de aplicații.
- Este **compilat și interpretat**, aceasta fiind soluția eficientă pentru obținerea portabilității.
- **Performanță** - deși mai lent decât limbajele de programare care generează executabile native pentru o anumită platformă de lucru, compilatorul Java asigură o performanță ridicată a codului de octeți, astfel încât viteza de lucru puțin mai scăzută nu va fi un impediment în dezvoltarea de aplicații oricât de complexe, inclusiv grafică 3D, animație, etc.
- Este **modelat după C și C++**, trecerea de la C, C++ la Java făcându-se foarte ușor.

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 așa numitele *platforme de lucru*, ce reprezintă seturi de librării scrise în limbajul Java, precum și diverse programe utilitare, folosite pentru dezvoltarea de aplicații sau componente destinate unei anume categorii de utilizatori.

- **J2SE** (Standard Edition)

Este platforma standard de lucru ce oferă suport pentru crearea de aplicații independente și appleturi.

De asemenea, aici este inclusă și tehnologia **Java Web Start** ce furnizează o modalitate extrem de facilă pentru lansarea și instalarea locală a programelor scrise în Java direct de pe Web, oferind cea mai comodă soluție pentru distribuția și actualizarea aplicațiilor 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 aplicații complexe, formate din componente ce trebuie să ruleze în sisteme eterogene, cu informațiile memorate în baze de date distribuite, etc.

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

Toate distribuțiile Java sunt oferite **gratuit** și pot fi descărcate de pe Internet de la adresa ”<http://java.sun.com>”.

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

1.1.3 Java: un limbaj compilat și interpretat

În funcție de modul de execuție a aplicațiilor, limbajele de programare se împart în două categorii:

- **Interpretate:** instrucțiunile sunt citite linie cu linie de un program numit *interpretor* și traduse în instrucțiuni mașină. Avantajul acestei soluții este simplitatea și faptul că fiind interpretată direct sursa programului obținem portabilitatea. Dezavantajul evident este viteza de execuție redusă. Probabil cel mai cunoscut limbaj interpretat este limbajul Basic.
- **Compile:** codul sursă al programelor este transformat de *compilator* într-un cod ce poate fi executat direct de procesor, numit *cod*

mașină. Avantajul este execuția extrem de rapidă, dezavantajul fiind lipsa portabilității, 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 soluțiile amintite mai sus, programele Java fiind atât interpretate cât și compilate. Așadar vom avea la dispoziție un compilator responsabil cu transformarea surselor programului în așa numitul *cod de octeți*, precum și un interpretor ce va executa respectivul cod de octeți.

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

Codurile de octeți sunt seturi de instrucțiuni care seamănă cu codul scris în limbaj de asamblare și sunt generate de compilator independent de mediul de lucru. În timp ce codul mașină este executat direct de către procesor și poate fi folosit numai pe platforma pe care a fost creat, codul de octeți este interpretat de mediul Java și de aceea poate fi rulat pe orice platformă pe care este instalată mediul de execuție Java.

Prin *mașina virtuală Java (JVM)* vom înțelege mediul de execuție al aplicațiilor Java. Pentru ca un cod de octeți să poată fi executat pe un anumit calculator, pe acesta trebuie să fie instalată o mașină virtuală Java. Acest lucru este realizat automat de către distribuția J2SDK.

1.2 Primul program

Crearea oricărei aplicații Java presupune efectuarea următorilor pași:

1. Scriererea codului sursă

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

Toate aplicațiile Java conțin o clasă principală(primară) în care trebuie să se găsească metoda `main`. Clasele aplicației se pot găsi fie într-un singur fișier, fie în mai multe.

2. Salvarea fișierelor sursă

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

3. Compilarea aplicației

Pentru compilare vom folosi compilatorul `javac` din distribuția J2SDK. Apelul compilatorului se face pentru fișierul ce conține clasa principală a aplicației sau pentru orice fișier/fișiere cu extensia `java`. Compilatorul creează câte un fișier separat pentru fiecare clasă a programului. Acestea au extensia `.class` și implicit sunt plasate în același director cu fișierele sursă.

```
javac FirstApp.java
```

În cazul în care compilarea a reușit va fi generat fișierul `FirstApp.class`.

4. Rularea aplicației

Se face cu interpretorul `java`, apelat pentru unitatea de compilare core-spunzătoare clasei principale. Deoarece interpretorul are ca argument de intrare numele clasei principale și nu numele unui fișier, ne vom poziționa în directorul ce conține fișierul `FirstApp.class` și vom apela interpretorul astfel:

```
java FirstApp
```

Rularea unei aplicații care nu folosește interfață grafică, se va face într-o fereastră sistem.

Atenție

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

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 internațional care înlocuiește vechiul set de caractere ASCII și care folosește pentru reprezentarea caracterelor 2 octeți, 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 grecești $\alpha - \omega$
- `\u2200` - `\u22FF` : simboluri matematice ($\forall, \exists, \emptyset$, etc.)
- `\u4e00` - `\u9fff` : litere din alfabetul Han (Chinez, Japonez, Coreean)

Mai multe informații legate de reprezentarea Unicode pot fi obținute la adresa "<http://www.unicode.org>".

1.3.2 Cuvinte cheie

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

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

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

Începând cu versiunea 1.5, mai există și cuvântul cheie `enum`.

1.3.3 Identificatori

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

1.3.4 Literalii

Literalii pot fi de următoarele tipuri:

- **Intregi**

Sunt acceptate 3 baze de numerație : baza 10, baza 16 (încep cu caracterele 0x) și baza 8 (încep cu cifra 0) și pot fi de două tipuri:

- **normali** - se reprezintă pe 4 octeți (32 biți)
- **lungi** - se reprezintă pe 8 octeți (64 biți) și se termină cu caracterul L (sau l).

- **Flotanți**

Pentru ca un literal să fie considerat flotant el trebuie să aibă cel puțin o zecimală după virgulă, să fie în notație exponențială sau să aibă sufixul `F` sau `f` pentru valorile normale - reprezentate pe 32 biți, respectiv `D` sau `d` pentru valorile duble - reprezentate pe 64 biți.

Exemple: `1.0`, `2e2`, `3f`, `4D`.

- **Logici**

Sunt reprezentați de `true` - valoarea logică de adevăr, respectiv `false` - valoarea logică de fals.

Atenție

Spre deosebire de C++, literalii întregi `1` și `0` nu mai au semnificația 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 secvență *escape* scrisă între apostrofuri. Secvențele *escape* permit specificarea caracterelor care nu au reprezentare grafică și reprezentarea unor caractere speciale precum backslash, apostrof, etc. Secvențele *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

- **Șiruri de caractere**

Un literal șir de caractere este format din zero sau mai multe caractere între ghilimele. Caracterele care formează șirul pot fi caractere grafice sau secvențe escape.

Dacă șirul este prea lung el poate fi scris ca o concatenare de subșiruri de dimensiune mai mică, concatenarea șirurilor realizându-se cu operatorul `+`, ca în exemplul: `"Ana " + " are " + " mere "`. Șirul vid este `""`.

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

1.3.5 Separatori

Un separator este un caracter care indică sfârșitul unei unități lexicale și începutul alteia. În Java separatorii sunt următorii: `() [] ; , . .`. Instrucțiunile unui program se separă cu punct și virgulă.

1.3.6 Operatori

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

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

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

```
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

În Java există trei feluri de comentarii:

- Comentarii pe mai multe linii, închise între `/*` și `*/`.
- Comentarii pe mai multe linii care țin de documentație, închise între `/**` și `*/`. Textul dintre cele două secvențe este automat mutat în documentația aplicației de către generatorul automat de documentație **javadoc**.
- Comentarii pe o singură linie, care încep cu `//`.

Observații:

- Nu putem scrie comentarii în interiorul altor comentarii.
- Nu putem introduce comentarii în interiorul literalilor caracter sau șir de caractere.
- Secvențele `/*` și `*/` pot să apară pe o linie după secvența `//` dar își pierd semnificația. La fel se întâmplă cu secvența `//` în comentarii care încep cu `/*` sau `*/`.

1.4 Tipuri de date și variabile

1.4.1 Tipuri de date

În Java tipurile de date se împart în două categorii: **tipuri primitive** și **tipuri referință**. Java pornește de la premiza că ”orice este un obiect”, prin urmare tipurile de date ar trebui să fie de fapt definite de clase și toate variabilele ar trebui să memoreze instanțe ale acestor clase (obiecte). În principiu acest lucru este adevărat, însă, pentru ușurința programării, mai există și așa 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 octeți)
- **logic**: `boolean` (`true` și `false`)

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

Vectorii, clasele și interfețele sunt tipuri referință. Valoarea unei variabile de acest tip este, spre deosebire de tipurile primitive, o referință (adresă de memorie) către valoarea sau mulțimea 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` și `union`. Pointerii au fost eliminați din cauză că erau o sursă constantă de erori, locul lor fiind luat de tipul referință, iar `struct` și `union` nu își 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 referințe la obiecte (tip referință). Indiferent de tipul lor, pentru a putea fi folosite variabilele trebuie declarate și, eventual, inițializate.

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

Evident, există posibilitatea de a declara și inițializa mai multe variabile sau constante de același tip într-o singură instrucțiune astfel:

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

Convenția 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 mulți atomi lexicali, atunci primele litere ale celorlalți 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";
```

În funcție de locul în care sunt declarate variabilele se împart în următoarele categorii:

- a. Variabile membre, declarate în interiorul unei clase, vizibile pentru toate metodele clasei respective cât și pentru alte clase în funcție 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 excepțiilor (vezi "Tratarea excepțiilor").

```
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 execuției

Instrucțiunile Java pentru controlul execuției sunt foarte asemănătoare celor din limbajul C și pot fi împărțite în următoarele categorii:

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

1.5.1 Instrucțiuni 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 instrucțiunea `switch` nu pot fi decât de tipuri primitive.

1.5.2 Instrucțiuni 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 inițializare cât și în pasul de iterație pot fi mai multe instrucțiuni despărțite prin virgulă.

while

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

do-while

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

1.5.3 Instrucțiuni pentru tratarea excepțiilor

Instrucțiunile pentru tratarea excepțiilor sunt `try-catch-finally`, respectiv `throw` și vor fi tratate în capitolul "Excepții".

1.5.4 Alte instrucțiuni

- `break`: părăsește forțat corpul unei structuri repetitive.
- `continue`: termina forțat iterația curentă a unui ciclu și trece la următoarea iterație.
- `return [valoare]`: termină o metodă și, eventual, returnează o valoare.
- `numeEticheta:` : Definește o etichetă.

Deși în Java nu există `goto`, se pot defini totuși etichete folosite în expresii de genul: `break numeEticheta` sau `continue numeEticheta`, utile pentru a controla punctul de ieșire 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, sa-l declarăm. Acest lucru se face prin expresii de forma:

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

ca în exemplele de mai jos:

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

- **Instanțierea**

Declararea unui vector nu implică și alocarea memoriei necesare pentru reținerea elementelor. Operațiunea de alocare a memoriei, numită și instanțierea vectorului, se realizează întotdeauna prin intermediul operatorului `new`. Instanțierea 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. În urma instanțierii vor fi alocați: *nrElemente* * *dimensiune(Tip)* octeți necesari memorării elementelor din vector, unde prin *dimensiune(Tip)* am notat numărul de octeți 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 și instanțierea unui vector pot fi făcute simultan astfel:

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

- **Inițializarea** (opțional) După declararea unui vector, acesta poate fi inițializat, adică elementele sale pot primi niște valori inițiale, evident dacă este cazul pentru așa ceva. În acest caz instanțierea nu mai trebuie făcută explicit, alocarea memoriei făcându-se automat în funcție de numărul de elemente cu care se inițializează vectorul.

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

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

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

1.6.2 Tablouri multidimensionale

În Java tablourile multidimensionale sunt de fapt vectori de vectori. De exemplu, crearea și instanțierea unei matrici vor fi realizate astfel:

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

`matrice[i]` este linia i a matricii și reprezintă un vector cu $nrColoane$ elemente iar `matrice[i][j]` este elementul de pe linia i și 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 înțelege modalitatea de folosire a lui `length` trebuie menționat că fiecare vector este de fapt o instanță a unei clase iar `length` este o variabilă publică a acelei clase, în care este reținut 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 semnificație decât copierea elementelor lui a în b și 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

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

- **sort** - sortează ascendent un vector, folosind un algoritm de tip *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 egalității valorilor a doi vectori (au aceleași număr de elemente și 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 și 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**, și vor fi studiați în capitolul "Colecții".

1.7 Șiruri de caractere

În Java, un șir 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 șir de caractere este constant (nu se dorește schimbarea conținutului să pe parcursul execuției programului) atunci el va fi declarat de tipul **String**, altfel va fi declarat de tip **StringBuffer**. Diferența principală între aceste clase este că **StringBuffer** pune la dispoziție metode pentru modificarea conținutului șirului, cum ar fi: **append**, **insert**, **delete**, **reverse**.

Uzual, cea mai folosită modalitate de a lucra cu șiruri este prin intermediul clasei **String**, care are și unele particularități față de restul claselor menite să simplifice cât mai mult folosirea șirurilor de caractere. Clasa **StringBuffer** va fi utilizată predominant în aplicații dedicate procesării textelor cum ar fi editoarele de texte.

Exemple echivalente de declarare a unui șir:

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

Observați prima variantă de declarare a șirului `s` din exemplul de mai sus - de altfel, cea mai folosită - care prezintă o particularitate a clasei **String** față de restul claselor Java referitoare la instanțierea obiectelor sale.

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

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

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

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

Pentru a lămuri puțin lucrurile, ceea ce execută compilatorul atunci când întâlnește o secvență de genul `x = "a" + 1 + "b"` este:

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

Atenție însă la ordinea de efectuare a operațiilor. Șirul `s=1+2+"a"+1+2` va avea valoarea `"3a12"`, primul `+` fiind operatorul matematic de adunare iar al doilea `+`, cel de concatenare a șirurilor.

1.8 Folosirea argumentelor de la linia de comandă

1.8.1 Transmiterea argumentelor

O aplicație 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 opțiuni legate de funcționarea aplicației sau să furnizeze anumite date inițiale programului.

Atenție

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 aplicații, fiind specificate după numele aplicației și separate prin spațiu. De exemplu, să presupunem că aplicația `Sortare` ordonează lexicografic (alfabetic) liniile unui fișier și primește ca argument de intrare numele fișierului pe care să îl sorteze. Pentru a ordona fișierul `"persoane.txt"`, aplicația va fi lansată astfel:

```
java Sortare persoane.txt
```

Așadar, formatul general pentru lansarea unei aplicații care primește argumente de la linia de comandă este:

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

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

1.8.2 Primirea argumentelor

În momentul lansării unei aplicații interpretorul parcurge linia de comandă cu care a fost lansată aplicația și, în cazul în care există, transmite programului argumentele specificate sub forma unui vector de șiruri. Acesta este primit de aplicație 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 conține toate argumentele transmise programului de la linia de comandă.

În cazul apelului `java Sortare persoane.txt` vectorul `args` va conține un singur element pe prima sa poziție: `args[0]="persoane.txt"`.

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

1.8. FOLOSIREA ARGUMENTELOR DE LA LINIA DE COMANDĂ 33

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

În cazul în care aplicația presupune existența unor argumente de la linia de comandă, însă acestea nu au fost transmise programului la lansarea sa, vor apărea excepții (erori) de tipul `ArrayIndexOutOfBoundsException`. Tratatrea acestor excepții este prezentată în capitolul "Excepții".

Din acest motiv, este necesar să testăm dacă programul a primit argumentele de la linia de comandă necesare pentru funcționarea sa și, în caz contrar, să afișeze un mesaj de avertizare sau să folosească niște 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 conține pe prima poziție numele aplicației, întrucât în Java numele aplicației este chiar numele clasei principale, adică a clasei în care se găsește metoda `main`.

Să considerăm în continuare un exemplu simplu în care se dorește afișarea 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 (aplicația a primit 2 argumente):

```

Hello
Java

```

Apelul `java Afisare "Hello Java"` va produce însă alt rezultat (aplicația 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 șiruri (obiecte de tip `String`). În cazul în care unele dintre acestea reprezintă valori numerice ele vor trebui convertite din șiruri în numere. Acest lucru se realizează cu metode de tipul `parseTipNumeric` aflate în clasa corespunzătoare tipului în care vrem să facem conversia: `Integer`, `Float`, `Double`, etc.

Să considerăm, de exemplu, că aplicația `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 `parseTipNumeric` pot produce excepții (erori) de tipul `NumberFormatException` în cazul în care șirul primit ca parametru nu reprezintă un număr de tipul respectiv. Tratarea acestor excepții este prezentată în capitolul "Excepții".

Capitolul 2

Obiecte și clase

2.1 Ciclul de viață al unui obiect

2.1.1 Crearea obiectelor

În Java, ca în orice limbaj de programare orientat-obiect, crearea obiectelor se realizează prin *instanțierea* unei clase și implică următoarele lucruri:

- **Declararea**

Presupune specificarea tipului acelui obiect, cu alte cuvinte specificarea clasei acestuia (vom vedea că tipul unui obiect poate fi și o interfață).

```
NumeClasa numeObiect;
```

- **Instanțierea**

Se realizează prin intermediul operatorului **new** și are ca efect crearea efectivă a obiectului cu alocarea spațiului de memorie corespunzător.

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

- **Inițializarea**

Se realizează prin intermediul constructorilor clasei respective. Inițializarea este de fapt parte integrantă a procesului de instanțiere, 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 și nu simpla specificare a numelui clasei.

Mai general, instanțierea și inițializarea apar sub forma:

Spațiul de memorie nu este pre-allocat

Declararea unui obiect nu implică sub nici o formă alocarea de spațiu 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 informații despre obiect, schimbarea stării sale sau executarea unor acțiuni. Aceste lucruri se realizează 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 construcții 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). În 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* și 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 limbaaje de programare impun ca programatorul să țină evidența obiectelor create și 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 funcționarea defectuoasă a programelor.

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

Un obiect este eliminat din memorie de procesul de colectare atunci când nu mai există nici o referință la acesta. Referințele (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 valoarea `null`.

Cum funcționează colectorul de gunoie ?

Colectorul de gunoie este un proces de prioritate scăzută care se execută periodic, scanează dinamic memoria ocupată de programul Java aflat în execuție și marchează acele obiecte care au referințe 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 mașinii virtuale Java să ”depună eforturi” în recuperarea memoriei ocupate de obiecte care nu mai sunt folosite, fără a forța însă pornirea procesului.

Finalizare

Înainte ca un obiect să fie eliminat din memorie, procesul `gc` dă acelui obiect posibilitatea ”să curețe după el”, apelând metoda de finalizare a obiectului respectiv. Uzual, în timpul finalizării un obiect își închide fișierele și socket-urile folosite, distruge referințele către alte obiecte (pentru a ușura sarcina collectorului de gunoaie), etc.

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

Atenție

Nu confundați 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 aplicație Java, cealaltă modalitate fiind prin intermediul interfețelor. Declararea unei clase respectă următorul format general:

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

Așadar, prima parte a declarației o ocupă modificatorii clasei. Aceștia sunt:

- **public**

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

- **abstract**

Declară o clasă abstractă (șablon). O clasă abstractă nu poate fi instanțiată, fiind folosită doar pentru a crea un model comun pentru o serie de subclase. (vezi "Clase și metode abstracte")

- **final**

Declară că respectiva clasă nu poate avea subclase. Declarația claselor finale are două scopuri:

- *securitate*: unele metode pot aștepta ca parametru un obiect al unei anumite clase și nu al unei subclase, dar tipul exact al unui obiect nu poate fi aflat cu exactitate decât în momentul execuției; 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 spirit orientat-obiect*: O clasă "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/și că implementează una sau mai multe interfețe, ale căror nume trebuie separate prin virgulă.

2.2.2 Extinderea claselor

Spre deosebire de alte limbaje de programare orientate-obiect, Java permite doar **moștenirea simplă**, ceea ce înseamnă că o clasă poate avea un singur părinte (superclasă). Evident, o clasă poate avea oricâți moștenitori (subclase), de unde rezultă că mulțimea tuturor claselor definite în Java poate fi văzută ca un arbore, rădăcina acestuia fiind clasa `Object`. Așadar, `Object` este singura clasă care nu are părinte, fiind foarte importantă în modul de lucru cu obiecte și 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 moștenește de la părintele său toate variabilele și metodele care nu sunt private.

2.2.3 Corpul unei clase

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

- Declararea și, eventual, inițializarea variabilelor de instanță și de clasă (cunoscute împreună ca *variabile membre*).
- Declararea și implementarea constructorilor.
- Declararea și implementarea metodelor de instanță și 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 făcută 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 același nume cu metodele clasei, care poate fi chiar numele clasei, fără a exista posibilitatea apariției vreunei ambiguități din punctul de vedere al compilatorului. Acest lucru este însă total nerecomandat dacă ne gândim din perspectiva lizibilității (clarității) codului, dovedind un stil ineficient de programare.

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

Atenție

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

2.2.4 Constructorii unei clase

Constructorii unei clase sunt metode speciale care au același nume cu cel al clasei, nu returnează nici o valoare și sunt folosiți pentru inițializarea obiectelor acelei clase în momentul instanțierii lor.

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

```

    }
}

```

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

Să considerăm ca exemplu declararea unei clase care descrie noțiunea de dreptunghi și 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 apelați automat la instanțierea unui obiect. În 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 implementați mai mulți constructori pentru o clasă, pentru a nu repeta secvențele de cod scrise deja la constructorii cu mai multe argumente (mai generali). Mai eficient, fără

a repeta aceleași secvențe de cod în toți constructorii (cum ar fi afișarea 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
    }
}
```

Atenție

Apelul explicit al unui constructor nu poate apărea decât într-un alt constructor și trebuie să fie prima instrucțiune din constructorul respectiv.

Constructorul implicit

Constructorii sunt apelați automat la instanțierea unui obiect. În cazul în care scriem o clasă care nu are declarat nici un constructor, sistemul îi creează automat un constructor implicit, care nu primește nici un argument și care nu face nimic. Deci prezența 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 declarații 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 instanțieri 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
```

În cazul moștenirii unei clase, instanțierea unui obiect din clasa extinsă implică instanțierea unui obiect din clasa părinte. Din acest motiv, fiecare constructor al clasei fiu va trebui să aibă un constructor cu aceeași semnătură 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` și cel implicit.

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

2.2.5 Declararea variabilelor

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

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

Variabilele membre ale unei clase se declară în corpul clasei și 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 instanță sau de clasă
- alți modificatori

Generic, o variabilă se declară astfel:

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

unde un modificador poate fi :

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

Exemple de declarații 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 specificați pentru o variabilă, alții decât cei de acces care sunt tratați într-o secțiune separată: "Specificatori de acces pentru membrii unei clase".

- **static**

Prezența lui declară că o variabilă este variabilă de clasă și nu de instanță. (vezi "Membri de instanță și 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 convenție, 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 și 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 facilitare avansată a limbajului Java.

2.2.6 this și super

Sunt variabile predefinite care fac referința, în cadrul unui obiect, la obiectul propriu-zis (**this**), respectiv la instanța 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 modificador 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 specificați pentru o metodă, alții decât cei de acces care sunt tratați într-o secțiune separată.

- **static**

Prezența lui declară că o metodă este de clasă și nu de instanță. (vezi "Membri de instanță și membri de clasă")

```

void metodaInstanta();
static void metodaClasa();

```

- **abstract**

Permite declararea metodelor abstracte. O metodă abstractă este o metodă care nu are implementare și trebuie obligatoriu să facă parte dintr-o clasă abstractă. (vezi "Clase și 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 consistența stării unui obiect. De exemplu, studenților unei universități trebuie să li se calculeze media finală, în funcție de notele obținute la examene, în aceeași 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**

În cazul în care avem o librărie importantă de funcții scrise în alt limbaj de programare, cum ar fi C, C++ și limbajul de asamblare, acestea pot fi refolosite din programele Java. Tehnologia care permite acest lucru se numește *JNI (Java Native Interface)* și permite asocierea dintre metode Java declarate cu **native** și metode native scrise în limbajele de programare menționate.

- **synchronized**

Este folosit în cazul în care se lucrează cu mai multe fire de execuție 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 execuție. (vezi "Fire de execuție")

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 referință la un obiect al unei clase. În 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 instrucțiunii `return`, care trebuie să apară în toate situațiile de terminare a funcției.

```
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
    }
}
```

În cazul în care în declarația funcției tipul returnat este un tip primitiv de date, valoarea returnată la terminarea funcției trebuie să aibă obligatoriu acel tip sau un subtip al său, altfel va fi furnizată o eroare la compilare. În 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 referință 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` și 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 numărul și tipul argumentelor primite de acea metodă. Tipul de date al unui argument poate fi orice tip valid al limbajului Java, atât tip primitiv cât și tip referință.

```
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 referințe la alte metode (funcții), însă pot fi trimise referințe la obiecte care să conțină implementarea acelor metode, pentru a fi apelate.

Pâna la apariția 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 numărului și tipurilor argumentelor. Vom analiza într-o secțiune separată modalitate de specificare a unui număr variabil de argumente pentru o metodă.

Numele argumentelor primite trebuie să difere între ele și 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 diferențierea 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;
    }
}
```

În Java argumentele sunt trimise **doar prin valoare** (pass-by-value). Acest lucru înseamnă că metoda recepționează 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 aceeași valoare ca înaintea apelului, modificările făcute în cadrul metodei fiind pierdute.

Când argumentul este de tip referință, metoda nu poate schimba valoarea referinței obiectului, însă poate apela metodele acelui obiect și poate modifica orice variabilă membră accesibilă.

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

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 primește doar valorile variabilelor `valx`, `valy` și `valr` și nu referințe la ele (adresele lor de memorie), astfel încât să le poată modifica valorile. În 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 referință și, deși nu îi schimbăm valoarea (valoarea sa este adresa de memorie la care se găsește și nu poate fi schimbată),

putem schimba starea obiectului, adică informația propriu-zisă conținută 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* însoțite 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 și instanțiat cu un număr variabil de argumente, în funcție de apelul metodei. Tipul argumentelor poate fi referință sau primitiv. Metoda de mai jos afișează 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 și supradefinirea metodelor

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

- *supraîncărcarea (overloading)* : în cadrul unei clase pot exista metode cu același nume cu condiția ca signaturile lor să fie diferite (lista de argumente primite să difere fie prin numărul argumentelor, fie prin tipul lor) astfel încât la apelul funcției 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 același nume și aceeași semnătură 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 și funcția 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). În cazul în care o clasă nu supradefinește toate metodele abstracte ale părintelui, ea însăși este abstractă și va trebui declarată ca atare.

În 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 și metodele unei clase sunt: **public**, **protected**, **private** și cel implicit (la nivel de pachet), iar nivelul lor de acces este dat în tabelul de mai jos:

Specificator	Clasa	Subclasa	Pachet	Oriunde
private	X			
protected	X	X*	X	
public	X	X	X	X
implicit	X		X	

Așadar, dacă nu este specificat nici un modificador de acces, implicit nivelul de acces este la nivelul pachetului. În cazul în care declarăm un membru "protected" atunci accesul la acel membru este permis din subclasele clasei în care a fost declarat dar depinde și de pachetul în care se găsește subclasa: dacă sunt în același pachet accesul este permis, dacă nu sunt în același pachet accesul nu este permis decât pentru obiecte de tipul subclasei.

Exemple de declarații:

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

2.5 Membri de instanță și membri de clasă

O clasă Java poate conține două tipuri de variabile și metode :

- *de instanță*: declarate **fără** modificadorul `static`, specifice fiecărei instanțe create dintr-o clasă și
- *de clasă*: declarate **cu** modificadorul `static`, specifice clasei.

2.5.1 Variabile de instanță și de clasă

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

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

se declară de fapt o variabilă de instanță, 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

```

Așadar, fiecare obiect nou creat va putea memora valori diferite pentru variabilele sale de instanță.

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

NumeClasa.numeVariabilaStatice

```

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

```

Inițializarea variabilelor de clasă se face o singură dată, la încărcarea în memorie a clasei respective, și este realizată prin atribuiri obișnuite:

```

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

```

2.5.2 Metode de instanță și de clasă

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

- metodele de instanță operează atât pe variabilele de instanță cât și 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 și la variabilele statice, întrucât metodele de clasă nu depind de starea obiectelor clasei respective, apelul lor se poate face și sub forma:

`NumeClasa.numeMetodaStatica`

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

Metodele de instanță 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 folosiți pentru a pune la dispoziție valori și metode independente de starea obiectelor dintr-o anumită clasă.

Declararea eficientă a constantelor

Să considerăm situația când dorim să declarăm o constantă.

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

La fiecare instanțiere 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 aceleași valori pentru toate instanțele clasei. Declararea corectă a constantelor trebuie așadar făcută cu modificatorii `static` și `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 și este utilă în situațiile când trebuie să controlăm diverși parametri legați de crearea obiectelor unei clase.

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

Implementarea funcțiilor globale

Spre deosebire de limbajele de programare procedurale, în Java nu putem avea funcții globale definite ca atare, întrucât ”orice este un obiect”. Din acest motiv chiar și metodele care au o funcționalitate 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 funcția `sqrt` care extrage radicalul unui număr și care se găsește în clasa `Math`. Dacă nu ar fi fost funcție 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)` .

Așadar, funcțiile globale necesare unei aplicații vor fi grupate corespunzător în diverse clase și implementate ca metode statice.

2.5.4 Blocuri statice de inițializare

Variabilele statice ale unei clase sunt inițializate la un moment care precede prima utilizare activă a clasei respective. Momentul efectiv depinde de implementarea mașinii virtuale Java și poartă numele de *inițializarea clasei*. Pe lângă setarea valorilor variabilelor statice, în această etapă sunt executate și blocurile statice de inițializare ale clasei. Acestea sunt secvențe de cod de forma:

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

care se comportă ca o metodă statică apelată automat de către mașina virtuală. Variabilele referite într-un bloc static de inițializare 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 definiție, o clasă membră a unei alte clase, numită și *clasă de acoperire*. În funcție de situație, 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 și 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 față de celelalte clase și anume acces nerestricționat la **toate** variabilele clasei de acoperire, chiar dacă acestea sunt private. O clasă declarată locală unei metode va avea acces și 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`

Așadar, 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 membră a clasei. Pentru clasele imbricate locale unei metode nu sunt permisi acești modificatori.

Toate clasele imbricate pot fi declarate folosind modificatorii **abstract** și **final**, semnificația lor fiind aceeași ca și în cazul claselor obișnuite.

2.6.2 Clase interne

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

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

Diferențierea acestor denumiri se face deoarece:

- o "clasă imbricată" reflectă relația sintactică a două clase: codul unei clase apare în interiorul codului altei clase;
- o "clasă internă" reflectă relația dintre instanțele a două clase, în sensul că o instanța a unei clase interne nu poate exista decât în cadrul unei instanțe a clasei de acoperire.

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

Așadar, o clasă internă este o clasă imbricată ale cărei instanțe nu pot exista decât în cadrul instanțelor clasei de acoperire și care are acces direct la toți membrii clasei sale de acoperire.

2.6.3 Identificare claselor imbricate

După cum știm orice clasă produce la compilare așa numitele "unități de compilare", care sunt fișiere având numele clasei respective și extensia `.class` și care conțin toate informațiile despre clasa respectivă. Pentru clasele imbricate aceste unități 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 fișiere:

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

În cazul în care clasele imbricate au la rândul lor alte clase imbricate (situație mai puțin uzuală) denumirea lor se face după aceeași regulă: adăugarea unui '\$' și apoi numele clasei imbricate.

2.6.4 Clase anonime

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

Exemple de folosire a claselor anonime vor fi date în capitolul "Interfețe", precum și extensiv în capitolul "Interfața grafică cu utilizatorul".

Fișierele 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 și metode abstracte

Uneori în proiectarea unei aplicații este necesar să reprezentăm cu ajutorul claselor concepte abstracte care să nu poată fi instanțiate și 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". Într-un program nu avem însă nevoie de numere generice ci de numere de un anumit tip: întregi, reale, etc. Clasa `Number` servește ca superclasă pentru clasele concrete `Byte`, `Double`, `Float`, `Integer`, `Long` și `Short`, ce implementează obiecte pentru descrierea numerelor de un anumit tip. Așadar, clasa `Number` reprezintă un concept abstract și nu vom putea instanția 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**, combinația **abstract final** fiind semnalată ca eroare la compilare - de altfel, o clasă declarată astfel nu ar avea nici o utilitate.

O clasă abstractă poate conține aceleași elemente membre ca o clasă obișnuită, la care se adaugă declarații de metode abstracte - fără nici o implementare.

2.7.2 Metode abstracte

Spre deosebire de clasele obișnuite care trebuie să furnizeze implementări pentru toate metodele declarate, o clasă abstractă poate conține metode fără nici o implementare. Metodele fara nici o implementare se numesc *metode abstracte* și pot apărea doar în clase abstracte. În fața 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
}
```

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

fiecărei subclase în parte.

Un exemplu elocvent de folosire a claselor și metodelor abstracte este descrierea obiectelor grafice într-o manieră orientată-obiect.

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

Pentru a folosi stările și 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 obișnuite 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 menționate 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.

În 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 referință numerice
- **Reader, Writer**: superclasele abstracte ale fluxurilor de intrare/ieșire pe caractere
- **InputStream, OutputStream**: superclasele abstracte ale fluxurilor de intrare/ieșire pe octeți
- **AbstractList, AbstractSet, AbstractMap**: superclase abstracte pentru structuri de date de tip colecție

- **Component** : superclasa abstractă a componentelor folosite în dezvoltarea de aplicații cu interfață 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 secțiunea dedicată modalității de creare a unei clase, clauza "extends" specifică faptul că acea clasă extinde (moștenește) o altă clasă, numită superclasă. O clasă poate avea o singură superclasă (Java nu suportă moștenirea multiplă) și chiar dacă nu specificăm clauza "extends" la crearea unei clase ea totuși va avea o superclasă. Cu alte cuvinte, în Java orice clasă are o superclasă și numai una. Evident, trebuie să existe o excepție de la această regulă și anume clasa care reprezintă rădăcina ierarhiei formată de relațiile de moștenire dintre clase. Aceasta este clasa **Object**.

Clasa `Object` este și superclasa implicită a claselor care nu specifică o anumită superclasă. Declarațiile 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` definește și implementează comportamentul comun al tuturor celorlalte clase Java, cum ar fi:

- posibilitatea testării egalității valorilor obiectelor,
- specificarea unei reprezentări ca șir de caractere a unui obiect ,
- returnarea clasei din care face parte un obiect,
- notificarea altor obiecte că o variabilă de condiție 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 același tip și care să aibă aceeași stare (aceleași valori pentru variabilele sale).

- **equals, hashCode**

Acestea sunt, de obicei, supradefinite împreună. În metoda `equals` este scris codul pentru compararea egalității conținutului a două obiecte. Implicit (implementarea din clasa `Object`), această metodă compară referințele obiectelor. Uzual este redefinită pentru a testa dacă stările obiectelor coincid sau dacă doar o parte din variabilele lor coincid.

Metoda `hashCode` returnează un cod întreg pentru fiecare obiect, pentru a testa consistența obiectelor: același obiect trebuie să returneze același cod pe durata execuției programului.

Dacă două obiecte sunt egale conform metodei `equals`, atunci apelul metodei `hashCode` pentru fiecare din cele două obiecte ar trebui să returneze același întreg.

- **finalize**

În această metodă se scrie codul care ”curăță după un obiect” înainte de a fi eliminat din memorie de colectorul de gunoaie. (vezi ”Distrugearea obiectelor”)

- **toString**

Este folosită pentru a returna o reprezentare ca șir de caractere a unui obiect. Este utilă pentru concatenarea șirurilor cu diverse obiecte în vederea afișării, fiind apelată automat atunci când este necesară transformarea unui obiect în șir 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 parțial clasa numerelor complexe, și 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 împărțite în *primitive* și *referință*. Pentru fiecare tip primitiv există o clasă corespunzătoare care permite 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 inițializarea unui obiect având o anumită valoare primitivă și metode specializate pentru conversia unui obiect în tipul primitiv corespunzător, de genul *tipPrimitiveValue*:

```

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 primitive și referință sunt posibile, acest mecanism purtând numele de *autoboxing*, respectiv *auto-unboxing*. Conversia explicită va fi făcută 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**. Această soluție 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);
...
```

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

Capitolul 3

Excepții

3.1 Ce sunt excepțiile ?

Termenul *excepție* este o prescurtare pentru "eveniment excepțional" și poate fi definit ca un eveniment ce se produce în timpul execuției unui program și care provoacă întreruperea cursului normal al execuției acestuia.

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

În momentul când o asemenea eroare se produce în timpul execuției va fi generat un obiect de tip excepție ce conține:

- informații despre excepția respectivă;
- starea programului în momentul producerii acelei excepții.

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

La rularea programului va fi generată o excepție, programul se va opri la instrucțiunea care a cauzat excepția și se va afișa 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 excepție se numește *aruncarea unei excepții* ("throwing an exception"). În momentul în care o metodă generează (aruncă) o excepție sistemul de execuție este responsabil cu găsirea unei secvențe de cod dintr-o metodă care să o trateze. Căutarea se face recursiv, începând cu metoda care a generat excepția și mergând înapoi pe linia apelurilor către acea metodă.

Secvența de cod dintr-o metodă care tratează o anumită excepție se numește *analizor de excepție* ("exception handler") iar interceptarea și tratarea ei se numește *prinderea excepției* ("catch the exception"). Cu alte cuvinte, la apariția unei erori este "aruncată" o excepție iar cineva trebuie să o "prindă" pentru a o trata. Dacă sistemul nu găsește nici un analizor pentru o anumită excepție, atunci programul Java se oprește cu un mesaj de eroare (în cazul exemplului de mai sus mesajul "Aici nu se mai ajunge..." nu va fi afișat).

Atenție

În Java tratarea erorilor nu mai este o opțiune ci o constrângere. În aproape toate situațiile, o secvență de cod care poate provoca excepții trebuie să specifice modalitatea de tratare a acestora.

3.2 "Prinderea" și tratarea excepțiilor

Tratarea excepțiilor se realizează prin intermediul blocurilor de instrucțiuni `try`, `catch` și `finally`. O secvență de cod care tratează anumite excepții trebuie să arate astfel:

```
try {  
    // Instrucțiuni 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 fișier octet cu octet și afisarea lui pe ecran. Fără a folosi tratarea excepțiilor metoda responsabilă cu citirea fișierului 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("\n\nInchidem fisierul " + fis);
    f.close();
}
```

Această secvență de cod va furniza erori la compilare deoarece în Java tratarea erorilor este obligatorie. Folosind mecanismul excepțiilor metoda `citeste` își poate trata singură erorile care pot surveni pe parcursul execuției sale. Mai jos este codul complet și corect al unui program ce afișează pe ecran conținutul unui fișier al cărui nume este primit ca argument de la linia de comandă. Tratarea excepțiilor 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 instrucțiunile de deschidere a unui fișier și de citire dintr-un fișier, ambele putând produce excepții. Excepțiile provocate de aceste instrucțiuni sunt tratate în cele două blocuri "catch", câte unul pentru fiecare tip de excepție. Închiderea fișierului se face în blocul "finally", deoarece acesta este sigur că se va executa. Fără a folosi blocul "finally", închiderea fișierului ar fi trebuit făcută în fiecare situație în care fișierul 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 semnalată în această situație este faptul că metoda `close`, responsabilă cu închiderea unui fișier, poate provoca la rândul său excepții, de exemplu atunci când fișierul mai este folosit și de alt proces și nu poate fi închis. Deci, pentru a avea un cod complet corect trebuie să tratăm și posibilitatea apariției unei excepții la metoda `close`.

Atenție

Obligatoriu un bloc de instrucțiuni "try" trebuie să fie urmat de unul sau mai multe blocuri "catch", în funcție de excepțiile provocate de acele instrucțiuni sau (opțional) de un bloc "finally".

3.3 ”Aruncarea” excepțiilor

În cazul în care o metodă nu își asumă responsabilitatea tratării uneia sau mai multor excepții pe care le pot provoca anumite instrucțiuni din codul său atunci ea poate să ”arunce” aceste excepții către metodele care o apelează, urmând ca acestea să implementeze tratarea lor sau, la rândul lor, să ”arunce” mai departe excepțiile respective.

Acest lucru se realizează prin specificarea în declarația metodei a clauzei `throws`:

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

Atenție

O metodă care nu tratează o anumită excepție trebuie obligatoriu să o ”arunce”.

În exemplul de mai sus dacă nu facem tratarea excepțiilor 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!");
}
}
```

Observați că, în acest caz, nu mai putem diferenția excepțiile provocate de citirea din fișier și de închiderea fișierului, ambele fiind de tipul `IOException`. De asemenea, închiderea fișierului nu va mai fi făcută în situația în care apare o excepție la citirea din fișier. Este situația 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 excepțiile mai departe către metoda care a apelat-o la rândul ei. Această înlănțuire se termină cu metoda `main` care, dacă va arunca excepțiile ce pot apărea în corpul ei, va determina trimiterea excepțiilor către mașina virtuală Java.

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

Tratarea excepțiilor de către JVM se face prin terminarea programului și afișarea informațiilor despre excepția 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 excepțiilor și aruncarea lor către nivelele superioare, astfel încât codul să fie cât mai clar și identificarea locului în care a apărut excepția să fie cât mai ușor de făcut.

Aruncarea unei excepții se poate face și implicit prin instrucțiunea `throw` ce are formatul: `throw excepție`, ca în exemplele de mai jos:

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

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

Această instrucțiune este folosită mai ales la aruncarea excepțiilor proprii. (vezi "Crearea propriilor excepții")

3.4 Avantajele tratării excepțiilor

Prin modalitatea sa de tratare a excepțiilor, Java are următoarele avantaje față de mecanismul tradițional 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 excepții corespunzător.
- Gruparea erorilor după tipul lor.

3.4.1 Separarea codului pentru tratarea erorilor

În programarea tradițională tratarea erorilor se combină cu codul ce poate produce apariția lor producând așa numitul "cod spaghetti". Să considerăm următorul exemplu: o funcție care încarcă un fișier în memorie:

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

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

- fișierul nu poate fi deschis
- nu se poate determina dimensiunea fișierului

- nu poate fi alocată suficientă memorie
- nu se poate face citirea din fișier
- fișierul nu poate fi închis

Un cod tradițional 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 programare este extrem de susceptibil la erori și îngreunează extrem de mult înțelegerea sa. În Java, folosind mecanismul excepțiilor, codul ar arăta, 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;}
}
```

Diferența de claritate este evidentă.

3.4.2 Propagarea erorilor

Propagarea unei erori se face până la un analizor de excepții corespunzător. Să presupunem că apelul la metoda `citesteFisier` este consecința 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. Tradițional, 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 excepțiile apărute în cadrul ei la un nivel superior, adică funcțiilor care o apelează sau sistemului. Cu alte cuvinte, o metodă poate să nu își asume responsabilitatea tratării excepțiilor 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

În Java există clase corespunzătoare tuturor excepțiilor care pot apărea la execuția unui program. Acestea sunt grupate în funcție de similaritățile lor într-o ierarhie de clase. De exemplu, clasa `IOException` se ocupă cu excepțiile ce pot apărea la operații de intrare/iesire și diferențiază la rândul ei alte tipuri de excepții, cum ar fi `FileNotFoundException`, `EOFException`, etc.

La rândul ei, clasa `IOException` se încadrează într-o categorie mai largă de excepții și anume clasa `Exception`.

Radacina acestei ierarhii este clasa `Throwable` (vezi "Ierarhia claselor ce descriu excepții").

Pronderea unei excepții se poate face fie la nivelul clasei specifice pentru acea excepție, fie la nivelul uneia din superclasele sale, în funcție de necesitățile programului, însă, cu cât clasa folosită este mai generică cu atât tratarea excepțiilor programul își 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
```

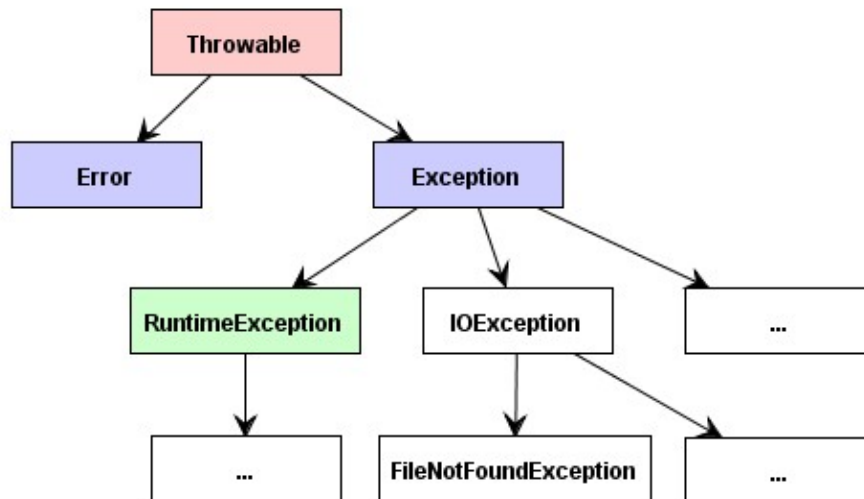
```

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 excepții

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



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

Excepțiile, obiectele de tip `Exception`, sunt excepțiile standard (soft) care trebuie tratate de către programele Java. După cum am mai zis tratarea acestor excepții nu este o opțiune ci o constrângere. Excepțiile care pot "scăpa" netratate descind din subclasa `RuntimeException` și se numesc *excepții la execuție*.

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

- `getMessage` - afișează detaliul unei excepții;
- `printStackTrace` - afișează informații complete despre excepție și localizarea ei;
- `toString` - metodă moștenită din clasa `Object`, care furnizează reprezentarea ca șir de caractere a excepției.

3.6 Excepții la execuție

În general, tratarea excepțiilor este obligatorie în Java. De la acest principiu se sustrag însă așa numitele *excepții la execuție* sau, cu alte cuvinte, excepțiile care provin strict din vina programatorului și nu generate de o anumită situație externă, cum ar fi lipsa unui fișier.

Aceste excepții au o superclasă comună `RuntimeException` și în acesata categorie sunt incluse excepțiile provocate de:

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

Excepțiile la execuție pot apărea oriunde în program și pot fi extrem de numeroare iar încercarea de "prindere" a lor ar fi extrem de anevoioasă. Din acest motiv, compilatorul permite ca aceste excepții să rămână netratate, tratarea lor nefiind însă ilegală. Reamintim însă că, în cazul apariției oricărui tip de excepție 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...");

```

Împărțirea la 0 va genera o excepție doar dacă tipul numerelor împărțite este aritmetic întreg. În cazul tipurilor reale (`float` și `double`) nu va fi generată nici o excepție, ci va fi furnizat ca rezultat o constantă care poate fi, funcție de operație, `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 excepții

Adeseori poate apărea necesitatea creării unor excepții proprii pentru a pune în evidența cazuri speciale de erori provocate de metodele claselor unei librării, cazuri care nu au fost prevăzute în ierarhia excepțiilor standard Java.

O excepție proprie trebuie să se încadreze însă în ierarhia excepțiilor Java, cu alte cuvinte clasa care o implementează trebuie să fie subclasă a uneia deja existente în aceasta ierarhie, preferabil una apropiată ca semnificație, 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 parțial o stivă de numere întregi cu operațiile de adăugare a unui element, respectiv de scoatere a elementului din vârful stivei. Dacă presupunem că stiva poate memora maxim 100 de elemente, ambele operații pot provoca excepții. Pentru a personaliza aceste excepții vom crea o clasă specifică denumită `ExceptieStiva`:

Listing 3.3: Excepții 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--];
    }
}
```

Secvența cheie este `extends Exception` care specifică faptul că noua clasă `ExceptieStiva` este subclasă a clasei `Exception` și deci implementează obiecte ce reprezintă excepții.

În general, codul adăugat claselor pentru excepții proprii este nesemnificativ: unul sau doi constructori care afișează un mesaj de eroare la ieșirea standard. Procesul de creare a unei noi excepții poate fi dus mai departe prin adăugarea unor noi metode clasei ce descrie acea excepție, însă aceasta dezvoltare nu își are rostul în majoritatea cazurilor. Excepțiile 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 și ieșiri

4.1 Introducere

4.1.1 Ce sunt fluxurile?

Majoritatea aplicațiilor necesită citirea unor informații care se găsesc pe o sursă externă sau trimiterea unor informații către o destinație externă. Informația se poate găsi oriunde: într-un fișier pe disc, în rețea, în memorie sau în alt program și poate fi de orice tip: date primitive, obiecte, imagini, sunete, etc. Pentru a aduce informații dintr-un mediu extern, un program Java trebuie să deschidă un *canal de comunicație (flux)* de la sursa informațiilor (fișier, memorie, socket, etc) și să citească secvențial informațiile respective.

Similar, un program poate trimite informații către o destinație externă deschizând un canal de comunicație (flux) către acea destinație și scriind secvențial informațiile respective.

Indiferent de tipul informațiilor, 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 și destinația lor sunt văzute ca fiind niște procese care produc, respectiv consumă informații.

Definiții:

Un *flux* este un canal de comunicație unidirecțional între două procese.

Un proces care descrie o sursă externă de date se numește *proces producător*.

Un proces care descrie o destinație externă pentru date se numește *proces consumator*.

Un flux care citește date se numește *flux de intrare*.

Un flux care scrie date se numește *flux de ieșire*.

Observații:

Fluxurile sunt canale de comunicație seriale pe 8 sau 16 biți.

Fluxurile sunt unidirecționale, de la producător la consumator.

Fiecare flux are un singur proces producător și un singur proces consumator.

Între două procese pot exista oricâte fluxuri, orice proces putând fi atât producător cât și consumator în același timp, dar pe fluxuri diferite.

Consumatorul și producătorul nu comunică direct printr-o interfață de flux ci prin intermediul codului Java de tratare a fluxurilor.

Clasele și interfețele standard pentru lucrul cu fluxuri se găsesc în pachetul `java.io`. Deci, orice program care necesită operații de intrare sau ieșire trebuie să conțină instrucțiunea de import a pachetului `java.io`:

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

4.1.2 Clasificarea fluxurilor

Există trei tipuri de clasificare a fluxurilor:

- După *direcția* canalului de comunicație deschis fluxurile se împart în:
 - fluxuri de intrare (pentru citirea datelor)
 - fluxuri de ieșire (pentru scrierea datelor)
- După *tipul de date* pe care operează:
 - fluxuri de octeți (comunicarea serială se realizează pe 8 biți)
 - fluxuri de caractere (comunicarea serială se realizează pe 16 biți)

- După *acțiunea* 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 și
- **Writer**- pentru fluxuri de ieșire.

Acestea sunt superclase abstracte pentru toate clasele ce implementează fluxuri specializate pentru citirea/scrierea datelor pe 16 biți și vor conține metodele comune tuturor.

Ca o regulă generală, toate clasele din aceste ierarhii vor avea terminația **Reader** sau **Writer** în funcție 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 *XReader* îi corespunde uzual un flux de ieșire *XWriter*, însă acest lucru nu este obligatoriu.

Clasele radacină pentru ierarhia fluxurilor de octeți sunt:

- **InputStream**- pentru fluxuri de intrare și
- **OutputStream**- pentru fluxuri de ieșire.

Acestea sunt superclase abstracte pentru clase ce implementează fluxuri specializate pentru citirea/scrierea datelor pe 8 biți. Ca și în cazul fluxurilor pe caractere denumirile claselor vor avea terminația superclasei lor: **FileInputStream**, **BufferedInputStream**, **FileOutputStream**, **BufferedOutputStream**, etc., fiecărui flux de intrare *XInputStream* corespunzându-i uzual un flux de ieșire *XOutputStream*, fără ca acest lucru să fie obligatoriu.

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

4.1.4 Metode comune fluxurilor

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

Reader	InputStream
<code>int read()</code>	<code>int read()</code>
<code>int read(char buf[])</code>	<code>int read(byte buf[])</code>
...	...

De asemenea, ambele clase pun la dispoziție metode pentru marcarea unei locații într-un flux, saltul peste un număr de poziții, resetarea poziției curente, etc. Acestea sunt însă mai rar folosite și nu vor fi detaliate.

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

Reader	InputStream
<code>void write(int c)</code>	<code>void write(int c)</code>
<code>void write(char buf[])</code>	<code>void write(byte buf[])</code>
<code>void write(String str)</code>	-
...	...

Inchiderea oricărui flux se realizează prin metoda `close`. În 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 referință la el, însă acest lucru trebuie evitat deoarece, la lucrul cu fluxului cu zonă tampon de memorie, datele din memorie vor fi pierdute la închiderea fluxului de către `gc`.

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

4.2 Folosirea fluxurilor

Așa cum am văzut, fluxurile pot fi împărțite în funcție de activitatea lor în fluxuri care se ocupă efectiv cu citirea/scrierea datelor și fluxuri pentru procesarea datelor (de filtrare). În continuare, vom vedea care sunt cele mai importante clase din cele două categorii și la ce folosesc acestea, precum și modalitățile de creare și utilizare a fluxurilor.

4.2.1 Fluxuri primitive

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

- **Fișier**

`FileReader`, `FileWriter`

`FileInputStream`, `FileOutputStream`

Numite și *fluxuri fișier*, acestea sunt folosite pentru citirea datelor dintr-un fișier, respectiv scrierea datelor într-un fișier și vor fi analizate într-o secțiune separată (vezi "Fluxuri pentru lucrul cu fișiere").

- **Memorie**

`CharArrayReader`, `CharArrayWriter`

`ByteArrayInputStream`, `ByteArrayOutputStream`

Aceste fluxuri folosesc pentru scrierea/citirea informațiilor în/din memorie și sunt create pe un vector existent deja. Cu alte cuvinte, permit tratarea vectorilor ca sursă/destinație pentru crearea unor fluxuri de intrare/ieșire.

`StringReader`, `StringWriter`

Permit tratarea șirurilor de caractere aflate în memorie ca sursă/destinație pentru crearea de fluxuri.

- **Pipe**

`PipedReader`, `PipedWriter`

`PipedInputStream`, `PipedOutputStream`

Implementează componentele de intrare/ieșire ale unei conducte de

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

4.2.2 Fluxuri de procesare

Fluxurile de procesare (sau de filtrare) sunt responsabile cu preluarea datelor de la un flux primitiv și 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` și să ofere informația dintr-un fișier 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 funcție de tipul de procesare pe care îl efectuează astfel:

- **”Bufferizare”**

`BufferedReader`, `BufferedWriter`

`BufferedInputStream`, `BufferedOutputStream`

Sunt folosite pentru a introduce un buffer în procesul de citire/scriere a informațiilor, reducând astfel numărul de accesări la dispozitivul ce reprezintă sursa/destinația originală a datelor. Sunt mult mai eficiente decât fluxurile fără buffer și din acest motiv se recomandă folosirea lor ori de câte ori este posibil (vezi ”Citirea și scrierea cu zona tampon”).

- **Filtrare**

`FilterReader`, `FilterWriter`

`FilterInputStream`, `FilterOutputStream`

Sunt clase abstracte ce definesc o interfață comună pentru fluxuri care filtrează automat datele citite sau scrise (vezi ”Fluxuri pentru filtrare”).

- **Conversie octeți-caractere**

`InputStreamReader`, `OutputStreamWriter`

Formează o punte de legătură între fluxurile de caractere și fluxurile de octeți. Un flux `InputStreamReader` citește octeți dintr-un flux `InputStream` și îi convertește la caractere, folosind codificarea standard a caracterelor sau o codificare specificată de program. Similar, un flux `OutputStreamWriter` convertește caractere în octeți și trimite rezultatul către un flux de tipul `OutputStream`.

- **Concatenare**
`SequenceInputStream`
Concatenează mai multe fluxuri de intrare într-unul singur (vezi "Concatenarea fișierelor").
- **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 mașina pe care se lucrează (vezi "Folosirea claselor `DataInputStream` și `DataOutputStream`").
- **Numărare**
`LineNumberReader`
`LineNumberInputStream`
Oferă și 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 și caracterul (octetul) care urmează celui curent citit.
- **Afișare**
`PrintWriter`
`PrintStream`
Oferă metode convenabile pentru afișarea informațiilor.

4.2.3 Crearea unui flux

Orice flux este un obiect al clasei ce implementează fluxul respectiv. Crearea unui flux se realizează așadar similar cu crearea obiectelor, prin instrucțiunea **new** și 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
FileOutputStream out = new FileOutputStream("fisier.dat");
```

Așadar, crearea unui flux primitiv de date care citește/scrie informații 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 referința 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);
```

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

```
FluxProcesare numeFlux = new FluxProcesare(fluxPrimitiv);
```

În 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 ușor de înțeles, întrucât operația lor de bază este citirea, respectiv scrierea unui caracter sau octet dintr-un sau într-un fișier 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 fișier. Acesta poate fi un șir de caractere, un obiect de tip `File` sau un obiect de tip `FileDescriptor` (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 fișierului. Aceștia pot provoca excepții de tipul `FileNotFoundException` în cazul în care fișierul 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 excepțiile de tipul `FileNotFoundException` sau de tipul superclasei `IOException`.

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

Listing 4.1: Copierea unui fișier

```
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();
        }
    }
}
```

În cazul în care vom lansa aplicația iar în directorul curent nu există un fișier cu numele "in.txt", va fi generată o excepție de tipul `FileNotFoundException`. Aceasta va fi prinsă de program deoarece, `IOException` este superclasă pentru `FileNotFoundException`. Dacă există fișierul "in.txt", aplicația va crea un nou fișier "out.txt" în care va fi copiat conținutul primului. Dacă există deja fișierul "out.txt" el va fi re-

scris. Dacă doream să facem operația de adăugare(append) și nu de rescriere pentru fișierul "out.txt" foloseam:

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

4.2.5 Citirea și 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 informațiilor, reducând astfel numărul de accesări ale dispozitivului ce reprezintă sursa/destinația atelor. Din acest motiv, sunt mult mai eficiente decât fluxurile fără buffer și din acest motiv se recomandă folosirea lor ori de câte ori este posibil.

Clasa `BufferedReader` citește în avans date și le memorează într-o zonă tampon. Atunci când se execută o operație de citire, caracterul va fi preluat din buffer. În cazul în care buffer-ul este gol, citirea se face direct din flux și, odată cu citirea caracterului, vor fi memorati în buffer și caracterele care îi urmează. Evident, `BufferedInputStream` funcționează după același principiu, singura diferență fiind faptul că sunt citați octeți.

Similar lucrează și clasele `BufferedWriter` și `BufferedOutputStream`. La operațiile de scriere datele scrise nu vor ajunge direct la destinație, ci vor fi memorate într-un buffer de o anumită dimensiune. Atunci când bufferul este plin, conținutul acestuia va fi transferat automat la destinație.

Fluxurile de citire/scriere cu buffer sunt fluxuri de procesare și 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 folosiți 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)

```

În cazul constructorilor în care dimensiunea buffer-ului nu este specificată, aceasta primește valoarea implicită de 512 octeți (caractere).

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

```

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

```

Metoda `readLine`

Este specifică fluxurilor de citire cu buffer și permite citirea linie cu linie a datelor de intrare. O linie reprezintă o succesiune de caractere terminată cu simbolul pentru sfârșit de linie, dependent de platforma de lucru. Acesta este reprezentat în Java prin secvența escape `'\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 aplicații 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 citește din primul flux de intrare specificat până când se ajunge la sfârșitul acestuia, după care primul flux de intrare este închis și se deschide automat următorul flux de intrare din care se vor citi în continuare datele, după care procesul se repetă până la terminarea tuturor fluxurilor de intrare.

Constructorii acestei clase sunt:

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

Primul construiește un flux secvențial dintr-o mulțime de fluxuri de intrare. Fiecare obiect în enumerarea primită ca parametru trebuie să fie de tipul **InputStream**.

Cel de-al doilea construiește un flux de intrare care combină doar două fluxuri *s1* și *s2*, primul flux citit fiind *s1*.

Exemplul cel mai elocvent de folosire a acestei clase este concatenarea a două sau mai multor fișiere:

Listing 4.2: Concatenarea a două fișiere

```
/* 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 fișiere există două variante:

- folosirea unei enumerări - primul constructor (vezi "Colecții");
- concatenarea pe rând a acestora folosind al 2-lea constructor; concatenarea a 3 fișiere 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 atașează 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 și
- **FilterOutputStream** - pentru filtrarea fluxurilor de ieșire.

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

```

DataInputStream, DataOutputStream
BufferedInputStream, BufferedOutputStream
LineNumberInputStream
PushbackInputStream
PrintStream

```

Observați că toate aceste clase descriu fluxuri de octeți.

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

Așadar, 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 și *fluxuri de procesare*. Din acest motiv, fluxurile de filtrare vor conține anumite metode specializate pentru citirea/scrierea datelor, altele decât cele comune tuturor fluxurilor. De exemplu, clasa `BufferedInputStream` pune la dispoziție metoda `readLine` pentru citirea unei linii din fluxul de intrare.

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

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

4.2.8 Clasele `DataInputStream` și `DataOutputStream`

Aceste clase oferă metode prin care un flux nu mai este văzut ca o însiruire de octeți, ci de date primitive. Prin urmare, vor furniza metode pentru citirea și scrierea datelor la nivel de tip primitiv și nu la nivel de octet. Clasele care oferă un astfel de suport implementează interfețele `DataInput`, respectiv `DataOutput`. Acestea definesc metodele pe care trebuie să le pună la dispoziție în vederea citirii/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 **readXXX** și **writeXXX**, specificate de interfețele **DataInput** și **DataOutput** și pot provoca excepții de tipul **IOException**. Denumirile lor sunt sugestive pentru tipul de date pe care îl prelucrează. mai puțin **readUTF** și **writeUTF** care se ocupă cu obiecte de tip **String**, fiind singurul tip referință permis de aceste clase.

Scrierea datelor folosind fluxuri de acest tip se face în format binar, ceea ce înseamnă că un fișier în care au fost scrise informații folosind metode **writeXXX** nu va putea fi citit decât prin metode **readXXX**.

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

4.3 Intrări și ieșiri formate

Incepând cu versiunea 1.5, limbajul Java pune la dispoziții modalități simplificate pentru afișarea formatată a unor informații, respectiv pentru citirea de date formate de la tastatură.

4.3.1 Intrări formate

Clasa **java.util.Scanner** oferă o soluție simplă pentru formatarea unor informații citite de pe un flux de intrare fie pe octeți, 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 Ieșiri formate

Clasele `PrintStream` și `PrintWriter` pun la dispoziție, pe lângă metodele `print`, `println` care ofereau posibilitatea de a afișa un șir de caractere, și metodele `format`, `printf` (echivalente) ce permit afișarea formatată a unor variabile.

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

Formatarea șirurilor de caractere se bazează pe clasa `java.util.Formatter`.

4.4 Fluxuri standard de intrare și ieșire

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

- o intrare standard
- o ieșire standard
- o ieșire standard pentru erori

În general, intrarea standard este tastatura iar ieșirea standard este ecranul.

Intrarea și ieșirea 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` și sunt:

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

4.4.1 Afisarea informațiilor pe ecran

Am văzut deja numeroase exemple de utilizare a fluxului standard de ieșire, el fiind folosit la afișarea 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 afișarea erorilor se folosește similar și apare uzual în secvențele de tratare a excepțiilor. Implicit, este același cu fluxul standard de ieșire.

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

Fluxurile de ieșire pot fi folosite așadar fără probleme deoarece tipul lor este `PrintStream`, clasă concretă pentru scrierea datelor. În 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 să-l folosim împreună 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 tastatură și 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("Introduceți o linie:");
String linie = stdin.readLine();
System.out.println(linie);
```

În exemplul următor este prezentat un program care afișează liniile introduse de la tastatură până în momentul în care se introduce linia "exit" sau o linie vidă și menționează dacă șirul 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 Redirecarea fluxurilor standard

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

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

Redirectarea ieșirii este utilă în special atunci când sunt afișate foarte multe date pe ecran. Putem redirecta afisarea către un fișier pe care să-l citim după execuția programului. Secvența clasică de redirectare a ieșirii este către un fișier este:

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

Redirectarea erorilor într-un fișier poate fi de asemenea utilă și 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 primește 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 fișier, redirectând intrarea standard către acel fișier. În momentul când testarea programului a luat sfârșit 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 și îl împarte în "atomi lexicali". Rezultatul va consta în faptul că în loc să se citească octeți sau caractere, se vor citi, pe rând, atomii lexicali ai fluxului respectiv. Printr-un *atom lexical* se înțelege în general:

- un identificator (un șir care nu este între ghilimele)
- un număr
- un șir de caractere
- un comentariu
- un separator

Atomii lexicali sunt despărțiți între ei de separatori. Implicit, acești separatori sunt cei obișnuiți: spațiu, tab, virgulă, punct și virgula, etc., însă pot fi schimbați prin diverse metode ale clasei.

Constructorii clasei sunt:

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

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

- **TT_EOF** - atom ce marchează sfârșitul fluxului
- **TT_EOL** - atom ce marchează sfârșitul 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 și scrie în variabilele **nval** sau **sval** valoarea corespunzătoare atomului.

Exemplul tipic de folosire a unui analizor lexical este citirea unei secvențe de numere și șiruri aflate într-un fișier sau primite de la tastatură:

Listing 4.5: Citirea unor atomi lexicali dintr-un fișier

```
/* 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
}
}
}

```

Așadar, modul de utilizare tipic pentru un analizor lexical este într-o buclă "while", în care se citesc atomii unul câte unul cu metoda `nextToken`, pâna se ajunge la sfârșitul fluxului (`TT_EOF`). În cadrul buclei "while" se determină tipul atomului curent curent (întors de metoda `nextToken`) și apoi se află valoarea numerică sau șirul de caractere corespunzător atomului respectiv.

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

4.5 Clasa `RandomAccessFile` (fișiere cu acces direct)

După cum am văzut, fluxurile sunt procese secvențiale de intrare/ieșire. Acestea sunt adecvate pentru lucrul cu medii secvențiale de memorare a datelor, cum ar fi banda magnetică sau pentru transmiterea informațiilor prin rețea, deși sunt foarte utile și pentru dispozitive în care informația poate fi accesată direct.

Clasa `RandomAccessFile` are următoarele caracteristici:

- permite accesul nesecvențial (direct) la conținutul unui fișier;
- este o clasă de sine stătătoare, subclasă directă a clasei `Object`;
- se găsește în pachetul `java.io`;

- implementează interfețele `DataInput` și `DataOutput`, ceea ce înseamnă ca sunt disponibile metode de tipul `readXXX`, `writeXXX`, întocmai ca la clasele `DataInputStream` și `DataOutputStream`;
- permite atât citirea cât și scriere din/in fișiere cu acces direct;
- permite specificarea modului de acces al unui fișier (read-only, read-write).

Constructorii acestei clase sunt:

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

unde *modAcces* poate fi:

- "r" - fișierul este deschis numai pentru citire (read-only)
- "rw" - fișierul este deschis pentru citire și scriere (read-write)

Exemple:

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

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

Clasa `RandomAccessFile` suportă noțiunea de *pointer de fișier*. Acesta este un indicator ce specifică poziția curentă în fișier. La deschiderea unui fișier pointerul are valoarea 0, indicând începutul fișierului. Apeluri la metodele de citire/scriere deplasează pointerul fișierului cu numărul de octeți citați sau scriși de metodele respective.

În plus față de metodele de tip `read` și `write` clasa pune la dispoziție și metode pentru controlul poziției pointerului de fișier. Acestea sunt:

- `skipBytes` - mută pointerul fișierului înainte cu un număr specificat de octeți
- `seek` - poziționează pointerul fișierului înaintea unui octet specificat
- `getFilePointer` - returnează poziția pointerului de fișier.

4.6 Clasa File

Clasa `File` nu se referă doar la un fișier ci poate reprezenta fie un fișier anume, fie multimea fișierelor dintr-un director.

Specificarea unui fișier/director se face prin specificarea căii absolute spre acel fișier sau a căii relative față de directorul curent. Acestea trebuie să respecte convențiile de specificare a căilor și numelor fișierelor de pe platforma de lucru.

Utilitatei clasei `File` constă în furnizarea unei modalități de a abstractiza dependențele căilor și numelor fișierelor față de mașina gazdă, precum și punerea la dispoziție a unor metode pentru lucrul cu fișiere și directoare la nivelul sistemului de operare.

Astfel, în această clasă vom găsi metode pentru testarea existenței, ștergerea, redenumirea unui fișier sau director, crearea unui director, listarea fișierelor dintr-un director, etc.

Trebuie menționat și faptul că majoritatea constructorilor fluxurilor care permit accesul la fișiere acceptă ca argument un obiect de tip `File` în locul unui șir ce reprezintă numele fișierului 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 și vor fi prezentate prin intermediul exemplului următor care listează fișierele și sub-directoarele unui director specificat și, pentru fiecare din ele afișează diverse informații:

Listing 4.6: Listarea conținutului 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

Interfețe

5.1 Introducere

5.1.1 Ce este o interfață ?

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

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

Definiție

O *interfață* este o colecție de metode fără implementare și declarații de constante.

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

5.2 Folosirea interfețelor

5.2.1 Definirea unei interfețe

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

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

O interfață poate avea un singur modificador și anume **public**. O interfață publică este accesibilă tuturor claselor, indiferent de pachetul din care face parte, implicit nivelul de acces fiind doar la nivelul pachetului din care face parte interfața.

O interfață poate extinde oricâte interfețe. Acestea se numesc *superinterfețe* și sunt separate prin virgulă. (vezi "Moștenirea multiplă prin intermediul interfețelor").

Corpul unei interfețe poate conține:

- **constante**: acestea pot fi sau nu declarate cu modificadorii **public**, **static** și **final** care sunt implicați, nici un alt modificador neputând apărea în declarația unei variabile dintr-o interfață. Constantele unei interfețe trebuie obligatoriu inițializate.

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

    int MAX;
    // Incorect, lipseste initializarea

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

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

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

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

Atenție

- Variabilele unei interfețe sunt implicit publice chiar dacă nu sunt declarate cu modificatorul `public`.
- Variabilele unei interfețe sunt implicit constante chiar dacă nu sunt declarate cu modificatorii `static` și `final`.
- Metodele unei interfețe sunt implicit publice chiar dacă nu sunt declarate cu modificatorul `public`.
- În variantele mai vechi de Java era permis și modificatorul `abstract` în declarația interfeței și în declarațiile metodelor, însă acest lucru nu mai este valabil, deoarece atât interfața cât și metodele sale nu pot fi altfel decât abstracte.

5.2.2 Implementarea unei interfețe

Implementarea uneia sau mai multor interfețe 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 interfețe sau poate să nu implementeze nici una.

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

Atenție

Modificarea unei interfețe implică modificarea tuturor claselor care implementează acea interfață.

O interfață nu este o clasă, dar orice referință de tip interfață poate primi ca valoare o referință la un obiect al unei clase ce implementează interfața respectivă. Din acest motiv, interfețele pot fi privite ca tipuri de date și vom spune adesea că un obiect are tipul X , unde X este o interfață, dacă acesta este o instanță a unei clase ce implementează interfața X .

Implementarea unei interfețe poate să fie și 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 noțiunea de stivă de obiecte. Obiectele de tip stivă, indiferent de implementarea lor, vor trebui să conțină metodele:

- `push` - adaugă un nou element în stivă
- `pop` - elimină elementul din vârful stivei
- `peek` - returnează varful stivei
- `empty` - testează dacă stiva este vidă
- `toString` - returnează conținutul stivei sub forma unui șir de caractere.

Din punctul de vedere al structurii interne, o stivă poate fi implementată folosind un vector sau o listă înlănțuită, ambele soluții având avantaje și dezavantaje. Prima soluție este mai simplă de înțeles, 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 interfețe. Vom vedea imediat avantajele acestei abordări.

Listing 5.1: Interfața 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 situațiile anormale care pot apărea atunci când încercăm să punem un element în stivă și nu este posibil din lipsă de memorie, sau încercăm să accesăm vârful stivei și aceasta este vidă, vom defini o excepție proprie *StackException*:

Listing 5.2: Tipul de excepție 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;
}
}
```

Remarcați că, deși în interfață metodele nu sunt declarate explicit cu modificatorul `public`, ele sunt totuși publice și trebuie declarate ca atare în clasă.

Trebuie remarcat și faptul că metoda `toString` este definită deja în clasa `Object`, deci clasa noastră o are deja implementată și nu am fi obținut nici o eroare la compilare dacă nu o implementam explicit. Ceea ce facem acum este de fapt supradefinirea ei.

O altă observație importantă se referă la faptul că trebuie să declarăm în cadrul interfeței și excepțiile aruncate de metode, ce trebuie obligatoriu tratate.

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

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 observație pe care o facem aici este că, deși metoda `push` din interfață declară aruncarea unor excepții de tipul `StackException`, nu este obligatoriu ca metoda din clasă să specifice și ea acest lucru, atât timp cât nu generează excepții de acel tip. Invers este însă obligatoriu.

În continuare este prezentată o mică aplicație demonstrativă care folosește tipul de date nou creat și 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();
        }
    }
}

```

Observați folosirea interfeței `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`.

Observație

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

5.3 Interfețe și clase abstracte

La prima vedere o interfață nu este altceva decât o clasă abstractă în care toate metodele sunt abstracte (nu au nici o implementare).

Așadar, o clasă abstractă nu ar putea înlocui o interfață ?

Răspunsul la întrebare depinde de situație, însă în general este 'Nu'.

Deosebirea constă în faptul că unele clase sunt forțate să extindă o anumită clasă (de exemplu orice applet trebuie să fie subclasa a clasei `Applet`) și nu ar mai putea să extindă o altă clasă, deoarece în Java nu există decât moștenire simplă. Fără folosirea interfețelor nu am putea forța clasa respectivă să respecte diverse tipuri de protocoale.

La nivel conceptual, diferența constă în:

- extinderea unei clase abstracte forțează o relație între clase;
- implementarea unei interfețe specifică doar necesitatea implementării unor anumite metode.

În multe situații interfețele și clasele abstracte sunt folosite împreună pentru a implementa cât mai flexibil și eficient o anumită ierarhie de clase. Un exemplu sugestiv este dat de clasele ce descriu colecții. Ca să particularizăm, există:

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

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

5.4 Moștenire multiplă prin interfețe

Interfețele nu au nici o implementare și nu pot fi instanțiate. Din acest motiv, nu reprezintă nici o problemă ca anumite clase să implementeze mai multe interfețe sau ca o interfață să extindă mai multe interfețe (să aibă mai multe superinterfețe)

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

O interfață mosteneste atât constantele cât și declarațiile de metode de la superinterfețele sale. O clasă moșteneste doar constantele unei interfețe și responsabilitatea implementării metodelor sale.

Să considerăm un exemplu de clasa care implementează mai multe interfețe:

```
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 interfață care extinde mai multe interfețe:

```
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 situații de ambiguitate, atunci când există constante sau metode cu aceleași nume în mai multe interfețe, însă acest lucru trebuie întotdeauna evitat, deoarece scrierea unui cod care poate fi confuz este un stil prost de programare. În cazul în care acest lucru se întâmplă, compilatorul nu va furniza eroare decât dacă se încearcă referirea constantelor ambigue fără a le prefixa cu numele interfeței sau dacă metodele cu același nume nu pot fi deosbite, cum ar fi situația când au aceeași 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 și interfețe:

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

5.5 Utilitatea interfețelor

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

- definirea unor similarități între clase independente fără a forța artificial o legătură între ele;
- asigură că toate clasele care implementează o interfață pun la dispoziție metodele specificate în interfață - de aici rezultă posibilitatea implementării unor clase prin mai multe modalități și 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 interfețe este implicit declarată cu `public`, `static` și `final`, interfețele reprezintă o metodă convenabilă de creare a unor grupuri de constante care să fie folosite global într-o aplicație:

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

Folosirea acestor constante se face prin expresii de genul `NumeInterfata.constantă`, 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-ziși, transmiterea metodelor ca parametri este realizată în Java prin intermediul interfețelor. Atunci când o metodă trebuie să primească ca argument de intrare o referință la o altă funcție necesară execuției sale, cunoscută doar la momentul execuției, atunci argumentul respectiv va fi declarat de tipul unei interfețe care conține metoda respectivă. La execuție metoda va putea primi ca parametru orice obiect ce implementează interfața respectivă și deci conține și codul funcției respective, aceasta urmând să fie apelată normal pentru a obține rezultatul dorit.

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

Primul exemplu se referă la explorarea nodurilor unui graf. În fiecare nod trebuie să se execute prelucrarea informației din nodul respectiv prin intermediul unei funcții primite ca parametru. Pentru aceasta, vom defini o interfață `Funcție` care va specifica metoda trimisă ca parametru.

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

class Graf {
    //...
    void explorare(Funcție 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 secțiunea următoare, întrucât face parte din API-ul standard Java și vor fi puse în evidență, prin intermediul său, și alte tehnici de programare.

5.6 Interfața FilenameFilter

Instanțele claselor ce implementează aceasta interfață sunt folosite pentru a crea filtre pentru fișiere și sunt primite ca argumente de metode care listează conținutul unui director, cum ar fi metoda `list` a clasei `File`.

Așadar, putem spune că metoda `list` primește ca argument o altă funcție care specifică dacă un fișier va fi returnat sau nu (criteriul de filtrare).

Interfața `FilenameFilter` are o singură metodă: **accept** care specifică criteriul de filtrare și anume, testează dacă numele fișierului primit ca parametru îndeplinește condițiile dorite de noi.

Definiția interfeței este:

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

Așadar, orice clasă de specificare a unui filtru care implementează interfața `FilenameFilter` trebuie să implementeze metoda `accept` a acestei interfețe. Aceste clase mai pot avea și alte metode, de exemplu un constructor care să primească criteriul de filtrare. În 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 fișierelor sunt:

- **endsWith** - testează dacă un șir are o anumită terminație
- **indexOf** - testează dacă un șir conține un anumit subșir, returnând poziția acestuia, sau 0 în caz contrar.

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

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

Observați că aici interfața 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 funcții (funcția de filtrare `accept`) ca argument al unei metode.

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

Listing 5.6: Listarea fișierelor 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

În cazul în care nu avem nevoie de clasa care reprezintă filtrul pentru listarea fișierelor dintr-un director decât o singură dată, pentru a evita crearea unei noi clase de sine stătătoare care să fie folosită pentru instanțierea unui singur obiect, putem folosi clasă internă anonimă, această situație 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();
        }
    }
}

```

```
}  
}
```

Așadar, o modalitate uzuală de folosire a claselor anonime pentru instanțierea unui obiect care trebuie să respecte o interfață este:

```
metoda(new Interfata() {  
    // Implementarea metodelor interfetei  
});
```

5.7 Compararea obiectelor

Am văzut în primul capitol că o soluție facilă și 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}
```

În cazul în care elementele din vector sunt de tip primitiv, ca în exemplul de mai sus, nu există nici o problemă în a determina ordinea firească a elementelor. Ce se întâmplă însă atunci când vectorul conține referințe la obiecte de un anumit tip? Să considerăm următorul exemplu, în care dorim să sortăm un vector format din instanțe 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 referință

```
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 execuția acestei aplicații va fi obținută o excepție, deoarece metoda `sort` nu știe care este ordinea naturală a obiectelor de tip `Persoana`. Va trebui, într-un fel sau altul, să specificăm acest lucru.

5.7.1 Interfața Comparable

Interfața `Comparable` impune o ordine totală asupra obiectelor unei clase ce o implementează. Această ordine se numește *ordinea naturală* a clasei și este specificată prin intermediul metodei `compareTo`. Definiția interfeței este:

```
public interface Comparable {
    int compareTo(Object o);
}
```

Așadar, o clasă ale cărei instanțe trebuie să fie comparabil va implementa metoda `compareTo` care trebuie să returneze:

- **o valoare strict negativă:** dacă obiectul curent (`this`) este mai *mic* decât obiectul primit ca argument;
- **zero:** dacă obiectul curent este *egal* decât obiectul primit ca argument;

- **o valoare strict pozitivă:** dacă obiectul curent este mai *mare* decât obiectul primit ca argument.

Reamintim că metoda `equals`, moștenită din `Object` de toate clasele, determină dacă două obiecte sunt egale (au aceeași valoare). Spunem că ordinea naturală a unei clase C este *consistentă* cu `equals` dacă și numai dacă `(e1.compareTo((Object)e2) == 0)` are aceeași valoare logică cu `e1.equals((Object)e2`, pentru orice $e1, e2$ instanțe ale lui C .

`null` nu este instanță a nici unei clase și `e.compareTo(null)` trebuie să arunce o excepție 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);
    }
}
```

Observați folosirea operatorului `instanceof`, care verifică dacă un obiect este instanță a unei anumite clase. Metoda `compareTo` va arunca o excepție de tipul `ClassCastException` dacă se încearcă compararea unui obiect de tip `Persoana` cu un obiect de alt tip. Metoda `equals` va returna, pur și simplu, `false`.

5.7.2 Interfața Comparator

În cazul în care dorim să sortăm elementele unui vector ce conține referințe după alt criteriu decât ordinea naturală a elementelor, avem nevoie de o altă soluție. 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.

Interfața `java.util.Comparator` conține metoda `compare`, care impune o ordine totală asupra elementelor unei colecții. Aceasta returnează un întreg cu aceeași semnificație ca la metoda `compareTo` a interfeței `Comparator` și are următoarea definiție: `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]);
  }
}

```

Observați cum compararea a două șiruri de caractere se face tot cu metoda `compareTo`, clasa `String` implemenând interfața `Comparable`.

5.8 Adaptori

În cazul în care o interfață conține mai multe metode și, la un moment dat, avem nevoie de un obiect care implementează interfața respectiv dar nu specifică cod decât pentru o singură metodă, el trebuie totuși să implementeze toate metodele interfeței, 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 interfața *X*. Soluția la această problemă este folosirea **adaptorilor**.

Definiție

Un *adaptor* este o clasă abstractă care implementează o anumită interfață fără a specifica cod nici unei metode a interfeței.

```
public abstract class XAdapter implements X {
    public void metoda_1() {}
    public void metoda_2() {}
    ...
    public void metoda_n() {}
}
```

În situația 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 ”Interfața grafică cu utilizatorul”.

Capitolul 6

Organizarea claselor

6.1 Pachete

Definiție

Un *pachet* este o colecție de clase și interfețe înrudite din punctul de vedere al funcționalității lor. Sunt folosite pentru găsirea și utilizarea mai ușoară a claselor, pentru a evita conflictele de nume și pentru a controla accesul la anumite clase. În alte limbaje de programare pachetele se mai numesc librării sau biblioteci.

6.1.1 Pachetele standard (J2SDK)

Platforma standard de lucru Java se bazează pe o serie de pachete cu ajutorul cărora se pot construi într-o manieră simplificată aplicațiile. Există deci un set de clase deja implementate care modelează structuri de date, algoritmi sau diverse noțiuni esențiale în dezvoltarea unui program. Cele mai importante pachete și suportul oferit de lor sunt:

- **java.lang** - clasele de bază ale limbajului Java
- **java.io** - intrări/ieșiri, lucrul cu fișiere
- **java.util** - clase și interfețe utile
- **java.applet** - dezvoltarea de appleturi

- **java.awt** - interfața grafică cu utilizatorul
- **java.awt.event** - mecanismele de tratare a evenimentelor generate de utilizator
- **java.beans** - scrierea de componente reutilizabile
- **java.net** - programare de rețea
- **java.sql** - lucrul cu baze de date
- **java.rmi** - execuție la distanță *Remote Message Interface*
- **java.security** - mecanisme de securitate: criptare, autentificare
- **java.math** - operații matematice cu numere mari
- **java.text** - lucrul cu texte, date și numere independent de limbă
- **java.lang.reflect** - introspecție
- **javax.swing** - interfața grafică cu utilizatorul, mult îmbogățită față de AWT.
- ...

6.1.2 Folosirea membrilor unui pachet

Conform specificațiilor de acces ale unei clase și ale membrilor ei, doar clasele publice și membrii declarați publici ai unei clase sunt accesibili în afara pachetului în care se găsesc. După cum am văzut în secțiunea "Specificatori de acces pentru membrii unei clase", accesul implicit în Java este la nivel de *pachet*.

Pentru a folosi o clasă publică dintr-un anumit pachet, sau pentru a apela o metodă publică a unei clase publice a unui pachet, există trei soluții:

- specificarea numelui complet al clasei
- importul clasei respective
- importul întregului pachet în care se găsește clasa.

Specificarea *numelui complet* al clasei se face prin prefixarea *numelui scurt* al clasei cu numele pachetului din care face parte: `numePachet.NumeClasa`.

```
Button          - numele scurt al clasei
java.awt        - pachetul din care face parte
java.awt.Button - numele complet al clasei
```

Această metodă este recomandată doar pentru cazul în care folosirea acelei clase se face o singură dată sau foarte rar.

De exemplu, ar fi extrem de neplăcut să scriem de fiecare dată când vrem să declarăm un obiect grafic secvențe de genul:

```
java.awt.Button b1 = new java.awt.Button("OK");
java.awt.Button b2 = new java.awt.Button("Cancel");
java.awt.TextField tf1 = new java.awt.TextField("Neplacut");
java.awt.TextField tf2 = new java.awt.TextField("Tot neplacut");
```

În aceste situații, vom importa în aplicația noastră clasa respectivă, sau întreg pachetul din care face parte. Acest lucru se realizează prin instrucțiunea **import**, care trebuie să apară la începutul fișierelor sursă, înainte de declararea vreunei clase sau interfețe.

6.1.3 Importul unei clase sau interfețe

Se face prin instrucțiunea **import** în care specificăm numele complet al clasei sau interfeței pe care dorim să o folosim dintr-un anumit pachet:

```
import numePachet.numeClasa;

//Pentru exemplul nostru:
import java.awt.Button;
import java.awt.TextField;
```

Din acest moment, vom putea folosi în clasele fișierului în care am plasat instrucțiunea de import numele scurt al claselor `Button` și `TextField`:

```
Button b1 = new Button("OK");
Button b2 = new Button("Cancel");
TextField tf1 = new TextField("Placut");
TextField tf2 = new TextField("Foarte placut");
```

Această abordare este eficientă și recomandată în cazul în care nu avem nevoie decât de câteva clase din pachetul respectiv. Dacă în exemplul nostru am avea nevoie și de clasele `Line`, `Point`, `Rectangle`, `Polygon`, ar trebui să avem câte o instrucțiune de import pentru fiecare dintre ele:

```
import java.awt.Button;
import java.awt.TextField;
import java.awt.Rectangle;
import java.awt.Line;
import java.awt.Point;
import java.awt.Polygon;
```

În această situație ar fi mai simplu să folosim importul la cerere din întregul pachet și nu al fiecărei clase în parte.

6.1.4 Importul la cerere dintr-un pachet

Importul la cerere dintr-un anumit pachet se face printr-o instrucțiune `import` în care specificăm numele pachetului ale cărui clase și interfețe dorim să le folosim, urmat de simbolul `*`. Se numește *import la cerere* deoarece încărcarea claselor se face dinamic, în momentul apelării lor.

```
import numePachet.*;

//Pentru exemplul nostru:
import java.awt.*;
```

Din acest moment, vom putea folosi în clasele fișierului în care am plasat instrucțiunea de import numele scurt al tuturor claselor pachetului importat:

```
Button b = new Button("OK");
Point p = new Point(0, 0);
```

Atenție

* nu are semnificația uzuală de la fișiere de wildcard (mască) și nu poate fi folosit decât ca atare. O expresie de genul `import java.awt.C*`; va produce o eroare de compilare.

În cazul în care sunt importate două sau mai multe pachete care conțin clase (interfețe) cu același nume, atunci referirea la ele trebuie făcută doar folosind numele complet, în caz contrar fiind semnalată o *ambiguitate* de către compilator.

```
import java.awt.*;
// Contine clasa List

import java.util.*;
// Contine interfata List

...
List x; //Declaratie ambigua

java.awt.List a = new java.awt.List(); //corect
java.util.List b = new ArrayList(); //corect
```

Sunt considerate importate automat, pentru orice fișier sursă, următoarele pachete:

- pachetul `java.lang`

```
import java.lang.*;
// Poate sau nu sa apara
// Mai bine nu...
```

- pachetul curent
- pachetul implicit (fără nume)

6.1.5 Importul static

Această facilitățe, introdusă începând cu versiunea 1.5, permite referirea constantelor statice ale unei clase fără a mai specifica numele complet al acesteia și este implementată prin adăugarea cuvântului cheie **static** după cel de `import`:

```
import static numePachet.NumeClasa.*;
```

Astfel, în loc să ne referim la constantele clasei cu expresii de tipul `NumeClasa.CONSTANTA`, putem folosi doar numele constantei.

```
// Inainte de versiuna 1.5
import java.awt.BorderLayout.*;
...
fereastră.add(new Button(), BorderLayout.CENTER);

// Incepend cu versiunea 1.5
import java.awt.BorderLayout.*;
import static java.awt.BorderLayout.*;
...
fereastră.add(new Button(), CENTER);
```

Atenție

Importul static nu importă decât constantele statice ale unei clase, nu și clasa în sine.

6.1.6 Crearea unui pachet

Toate clasele și interfețele Java aparțin la diverse pachete, grupate după funcționalitatea lor. După cum am văzut clasele de bază se găsesc în pachetul `java.lang`, clasele pentru intrări/ieșiri sunt în `java.io`, clasele pentru interfața grafică în `java.awt`, etc.

Crearea unui pachet se realizează prin scriere la începutul fișierelor sursă ce conțin clasele și interfețele pe care dorim să le grupăm într-un pachet a instrucțiunii: **package numePachet;**

Să considerăm un exemplu: presupunem că avem două fișiere sursă `Graf.java` și `Arbore.java`.

```
//Fișierul Graf.java
package grafuri;
class Graf {...}
class GrafPerfect extends Graf {...}

//Fișierul Arbore.java
package grafuri;
class Arbore {...}
```

```
class ArboreBinar extends Arbore {...}
```

Clasele `Graf`, `GrafPerfect`, `Arbore`, `ArboreBinar` vor face parte din același pachet `grafuri`.

Instrucțiunea `package` acționează asupra întregului fișier sursă la începutul căruia apare. Cu alte cuvinte nu putem specifica faptul că anumite clase dintr-un fișier sursă aparțin unui pachet, iar altele altui pachet.

Dacă nu este specificat un anumit pachet, clasele unui fișier sursă vor face parte din pachetul implicit (care nu are nici un nume). În general, pachetul implicit este format din toate clasele și interfețele directorului curent de lucru. Este recomandat însă ca toate clasele și interfețele să fie plasate în pachete, pachetul implicit fiind folosit doar pentru aplicații mici sau prototipuri.

6.1.7 Denumirea unui pachet

Există posibilitatea ca doi programatori care lucrează la un proiect comun să folosească același nume pentru unele din clasele lor. De asemenea, se poate ca una din clasele unei aplicații să aibă același nume cu o clasă a mediului Java. Acest lucru este posibil atât timp cât clasele cu același nume se găsesc în pachete diferite, ele fiind diferențiate prin prefixarea lor cu numele pachetelor.

Ce se întâmplă însă când doi programatori care lucrează la un proiect comun folosesc clase cu același nume, ce se găsesc în pachete cu același nume?

Pentru a evita acest lucru, companiile folosesc inversul domeniului lor Internet în denumirea pachetelor implementate în cadrul companiei, cum ar fi `ro.companie.numePachet`. În cadrul aceleiași companii, conflictele de nume vor fi rezolvate prin diverse convenții de uz intern, cum ar fi folosirea numelui de cont al programatorilor în denumirea pachetelor create de aceștia. De exemplu, programatorul cu numele Ion al companiei XSoft, având contul `ion@xsoft.ro`, își va prefixa pachetele cu `ro.xsoft.ion`, pentru a permite identificarea în mod unic a claselor sale, indiferent de contextul în care acestea vor fi integrate.

6.2 Organizarea fișierelor

6.2.1 Organizarea fișierelor sursă

Orice aplicație nebanală trebuie să fie construită folosind o organizare ierarhică a componentelor sale. Este recomandat ca strategia de organizare a fișierelor sursă să respecte următoarele convenții:

- Codul sursă al claselor și interfețelor să se găsească în fișiere ale căror nume să fie chiar numele lor scurt și care să aibă extensia `.java`.

Atenție

Este **obligatoriu** ca o clasă/interfață publică să se găsească într-un fișier având numele clasei(interfeței) și extensia `.java`, sau compilatorul va furniza o eroare. Din acest motiv, într-un fișier sursă nu pot exista două clase sau interfețe publice. Pentru clasele care nu sunt publice acest lucru nu este obligatoriu, ci doar recomandat. Într-un fișier sursă pot exista oricâte clase sau interfețe care nu sunt publice.

-
- Fișierele sursă trebuie să se găsească în directoare care să reflecte numele pachetelor în care se găsesc clasele și interfețele din acele fișiere. Cu alte cuvinte, un director va conține surse pentru clase și interfețe din același pachet iar numele directorului va fi chiar numele pachetului. Dacă numele pachetelor sunt formate din mai multe unități lexicale separate prin punct, atunci acestea trebuie de asemenea să corespundă unor directoare ce vor descrie calea spre fișierele sursă ale căror clase și interfețe fac parte din pachetele respective.

Vom clarifica modalitatea de organizare a fișierelor sursă ale unei aplicații printr-un exemplu concret. Să presupunem că dorim crearea unor componente care să reprezinte diverse noțiuni matematice din domenii diferite, cum ar fi geometrie, algebră, analiză, etc. Pentru a simplifica lucrurile, să presupunem că dorim să creăm clase care să descrie următoarele noțiuni: *poligon, cerc, poliedru, sferă, grup, funcție*.

O primă variantă ar fi să construim câte o clasă pentru fiecare și să le plasăm

în același director împreună cu un program care să le folosească, însă, având în vedere posibila extindere a aplicației cu noi reprezentări de noțiuni matematice, această abordare ar fi inefficientă.

O abordare elegantă ar fi aceea în care clasele care descriu noțiuni din același domeniu să se găsească în pachete separate și directoare separate. Ierarhia fișierelor sursa ar fi:

```
/matematica
  /surse
    /geometrie
      /plan
        Poligon.java
        Cerc.java
      /spatiu
        Poliedru.java
        Sfera.java
    /algebra
      Grup.java
    /analiza
      Functie.java

  Matematica.java
```

Clasele descrise în fișierele de mai sus trebuie declarate în pachete denumite corespunzător cu numele directoarelor în care se găsesc:

```
// Poligon.java
package geometrie.plan;
public class Poligon { . . . }

// Cerc.java
package geometrie.plan;
public class Cerc { . . . }

// Poliedru.java
package geometrie.spatiu;
public class Poliedru { . . . }
```

```
// Sfera.java
package geometrie.spatiu;
public class Sfera { . . . }

// Grup.java
package algebra;
public class Grup { . . . }

// Functie.java
package analiza;
public class Functie { . . . }
```

`Matematica.java` este clasa principală a aplicației. După cum se observă, numele lung al unei clase trebuie să descrie calea spre acea clasă în cadrul fișierelor sursă, relativ la directorul în care se găsește aplicația.

6.2.2 Organizarea unităților de compilare (.class)

În urma compilării fișierelor sursă vor fi generate unități de compilare pentru fiecare clasă și interfață din fișierele sursă. După cum știm acestea au extensia `.class` și numele scurt al clasei sau interfeței respective.

Spre deosebire de organizarea surselor, un fișier `.class` **trebuie** să se găsească într-o ierarhie de directoare care să reflecte numele pachetului din care face parte clasa respectivă.

Implicit, în urma compilării fișierele sursă și unitățile de compilare se găsesc în același director, însă ele pot fi apoi organizate separat. Este recomandat însă ca această separare să fie făcută automat la compilare.

Revenind la exemplul de mai sus, vom avea următoarea organizare:

```
/matematica
  /clase
    /geometrie
      /plan
        Poligon.class
        Cerc.class
    /spatiu
```

```

    Poliedru.class
    Sfera.class
/algebra
    Grup.class
/analiza
    Functie.class

Matematica.class

```

Crearea acestei structuri ierarhice este făcută automat de către compilator. În directorul aplicației (*matematica*) creăm subdirectorul *clase* și dăm comanda:

```

javac -sourcepath surse surse/Matematica.java -d clase
sau
javac -classpath surse surse/Matematica.java -d clase

```

Opțiunea **-d** specifică directorul rădăcină al ierarhiei de clase. În lipsa lui, fiecare unitate de compilare va fi plasată în același director cu fișierul său sursă.

Deoarece compilăm clasa principală a aplicației, vor fi compilate în cascadă toate clasele referite de aceasta, dar numai acestea. În cazul în care dorim să compilăm explicit toate fișierele java dintr-un anumit director, de exemplu *surse/geometrie/plan*, putem folosi expresia:

```

javac surse/geometrie/plan/*.java -d clase

```

6.2.3 Necesitatea organizării fișierelor

Organizarea fișierelor sursă este necesară deoarece în momentul când compilatorul întâlnește un nume de clasă el trebuie să poată identifica acea clasă, ceea ce înseamnă că trebuie să găsească fișierul sursă care o conține.

Similar, unitățile de compilare sunt organizate astfel pentru a da posibilitatea interpretorului să găsească și să încarce în memorie o anumită clasă în timpul execuției programului.

Însă această organizare nu este suficientă deoarece specifică numai partea finală din calea către fișierele *.java* și *.class*, de exemplu */matematica/clase/geometrie/plan/Poligon.class*. Pentru aceasta, atât la compilare cât și la interpretare trebuie specificată lista de directoare rădăcină

în care se găsesc fișierele aplicației. Această listă se numește *cale de cautare* (*classpath*).

Definiție

O *cale de căutare* este o listă de directoare sau arhive în care vor fi căutate fișierele necesare unei aplicații. Fiecare director din calea de cautare este directorul imediat superior structurii de directoare corespunzătoare numelor pachetelor în care se găsesc clasele din directorul respectiv, astfel încât compilatorul și interpretorul să poată construi calea completă spre clasele aplicației. Implicit, calea de căutare este formată doar din directorul curent.

Să considerăm clasa principală a aplicației `Matematica.java`:

```
import geometrie.plan.*;
import algebra.Grup;
import analiza.Functie;

public class Matematica {
    public static void main(String args[]) {
        Poligon a = new Poligon();
        geometrie.spatiu.Sfera = new geometrie.spatiu.Sfera();
        //...
    }
}
```

Identificarea unei clase referite în program se face în felul următor:

- La directoarele aflate în calea de căutare se adaugă subdirectoarele specificate în `import` sau în numele lung al clasei
- În directoarele formate este căutat un fișier cu numele clasei. În cazul în care nu este găsit nici unul sau sunt găsite mai multe va fi semnalată o eroare.

6.2.4 Setarea căii de căutare (CLASSPATH)

Setarea căii de căutare se poate face în două modalități:

- Setarea variabilei de mediu **CLASSPATH** - folosind această variantă toate aplicațiile Java de pe mașina respectivă vor căuta clasele necesare în directoarele specificate în variabila **CLASSPATH**.

UNIX:

```
SET CLASSPATH = cale1:cale2:...
```

DOS shell (Windows 95/NT/...):

```
SET CLASSPATH = cale1;cale2;...
```

- Folosirea opțiunii **-classpath** la compilarea și interpretarea programelor - directoarele specificate astfel vor fi valabile doar pentru comanda curentă:

```
javac - classpath <cale de cautare> <surse java>
java - classpath <cale de cautare> <clasa principala>
```

Lansarea în execuție a aplicației noastre, din directorul `matematica`, se va face astfel:

```
java -classpath clase Matematica
```

În concluzie, o organizare eficientă a fișierelor aplicației ar arăta astfel:

```
/matematica
 /surse
 /clase
 compile.bat
 (javac -sourcepath surse surse/Matematica.java -d clase)
 run.bat
 (java -classpath clase Matematica)
```

6.3 Arhive JAR

Fișierele **JAR** (Java Archive) sunt arhive în format ZIP folosite pentru împachetarea aplicațiilor Java. Ele pot fi folosite și pentru comprimări obișnuite, diferența față de o arhivă ZIP obișnuită fiind doar existența unui director denumit `META-INF`, ce conține diverse informații auxiliare legate de aplicația sau clasele arhivate.

Un fișier JAR poate fi creat folosind utilitarul **jar** aflat în distribuția J2SDK, sau metode ale claselor suport din pachetul **java.util.jar**.

Dintre beneficiile oferite de arhivele JAR amintim:

- portabilitate - este un format de arhivare independent de platformă;

- compresare - dimensiunea unei aplicații în forma sa finală este redusă;
- minimizarea timpului de încărcare a unui applet: dacă appletul (fișiere class, resurse, etc) este compresat într-o arhivă JAR, el poate fi încărcat într-o singură tranzacție HTTP, fără a fi deci nevoie de a deschide câte o conexiune nouă pentru fiecare fișier;
- securitate - arhivele JAR pot fi "semnate" electronic
- mecanismul pentru lucrul cu fișiere JAR este parte integrată a platformei Java.

6.3.1 Folosirea utilitarului jar

Arhivatorul *jar* se găsește în subdirectorul *bin* al directorului în care este instalat kitul J2SDK. Mai jos sunt prezentate pe scurt operațiile uzuale:

- Crearea unei arhive
`jar cf arhiva.jar fișier(e)-intrare`
- Vizualizare conținutului
`jar tf nume-arhiva`
- Extragerea conținutului
`jar xf arhiva.jar`
- Extragerea doar a unor fișiere
`jar xf arhiva.jar fișier(e)-arhivate`
- Executarea unei aplicații
`java -jar arhiva.jar`
- Deschiderea unui applet arhivat
`<applet code=A.class archive="arhiva.jar" ...>`

Exemple:

- Arhivarea a două fișiere class:
`jar cf classes.jar A.class B.class`
- arhivarea tuturor fișierelor din directorul curent:
`jar cvf allfiles.jar *`

6.3.2 Executarea aplicațiilor arhivate

Pentru a rula o aplicație împachetată într-o arhivă JAR trebuie să facem cunoscută interpretorului numele clasei principale a aplicației. Să considerăm următorul exemplu, în care dorim să arhivăm clasele aplicației descrise mai sus, în care clasa principală era `Matematica.java`. Din directorul `clase` vom lansa comanda:

```
jar cvf mate.jar geometrie analiza algebra Matematica.class
```

În urma acestei comenzi vom obține arhiva `mate.jar`. Dacă vom încerca să lansăm în execuție această arhivă prin comanda `java -jar mate.jar` vom obține următoarea eroare: "Failed to load Main-Class manifest from mate.jar". Aceasta înseamnă că în fișierul `Manifest.mf` ce se găsește în directorul `META-INF` trebuie să înregistrăm clasa principală a aplicației. Acest lucru îl vom face în doi pași:

- se creează un fișier cu un nume oarecare, de exemplu `manifest.txt`, în care vom scrie:
`Main-Class: Matematica`
- adăugăm această informație la fișierul `manifest` al arhivei `mate.jar`:
`jar uvfm mate.jar manifest.txt`

Ambele operații puteau fi executate într-un singur pas:

```
jar cvfm mate.jar manifest.txt  
geometrie analiza algebra Matematica.class
```

Pe sistemele Win32, platforma Java 2 va asocia extensiile `.jar` cu interpretorul Java, ceea ce înseamnă că făcând dublu-click pe o arhivă JAR va fi lansată în execuție aplicația împachetată în acea arhivă (dacă există o clasă principală).

Capitolul 7

Colecții

7.1 Introducere

O *colecție* este un obiect care grupează mai multe elemente într-o singură unitate. Prin intermediul colecțiilor vom avea acces la diferite tipuri de date cum ar fi vectori, liste înlănțuite, stive, mulțimi matematice, tabele de dispersie, etc. Colecțiile sunt folosite atât pentru memorarea și manipularea datelor, cât și pentru transmiterea unor informații de la o metodă la alta.

Tipul de date al elementelor dintr-o colecție este `Object`, ceea ce înseamnă că mulțimile reprezentate sunt eterogene, putând include obiecte de orice tip.

Incepând cu versiunea 1.2, în Java colecțiile sunt tratate într-o manieră unitară, fiind organizate într-o arhitectură foarte eficientă și flexibilă ce cuprinde:

- **Interfețe:** tipuri abstracte de date ce descriu colecțiile și permit utilizarea lor independent de detaliile implementărilor.
- **Implementări:** implementări concrete ale interfețelor ce descriu colecții. Aceste clase reprezintă *tipuri de date reutilizabile*.
- **Algoritmi:** metode care efectuează diverse operații utile cum ar fi căutarea sau sortarea, definite pentru obiecte ce implementează interfețele ce descriu colecții. Acești algoritmi se numesc și *polimorfici* deoarece pot fi folosiți pe implementări diferite ale unei colecții, reprezentând elementul de *funcționalitate reutilizabilă*.

Utilizarea colecțiilor oferă avantaje evidente în procesul de dezvoltare a unei aplicații. Cele mai importante sunt:

- **Reducerea efortului de programare:** prin punerea la dispoziția programatorului a unui set de tipuri de date și algoritmi ce modelează structuri și operații des folosite în aplicații.
- **Creșterea vitezei și calității programului:** implementările efective ale colecțiilor sunt de înaltă performanță și folosesc algoritmi cu timp de lucru optim.

Astfel, la scrierea unei aplicații putem să ne concentrăm eforturile asupra problemei în sine și nu asupra modului de reprezentare și manipulare a informațiilor.

7.2 Interfețe ce descriu colecții

Interfețele reprezintă nucleul mecanismului de lucru cu colecții, scopul lor fiind de a permite utilizarea structurilor de date independent de modul lor de implementare.

Collection modelează o colecție la nivelul cel mai general, descriind un grup de obiecte numite și *elementele* sale. Unele implementări ale acestei interfețe permit existența elementelor duplicate, altele nu. Unele au elementele ordonate, altele nu. Platforma Java nu oferă nici o implementare directă a acestei interfețe, ci există doar implementări ale unor subinterfețe mai concrete, cum ar fi `Set` sau `List`.

```
public interface Collection {  
  
    // Metode cu caracter general  
    int size();  
    boolean isEmpty();  
    void clear();  
    Iterator iterator();  
  
    // Operatii la nivel de element  
    boolean contains(Object element);  
    boolean add(Object element);  
    boolean remove(Object element);  
}
```

```

// Operatii la nivel de multime
boolean containsAll(Collection c);
boolean addAll(Collection c);
boolean removeAll(Collection c);
boolean retainAll(Collection c);

// Metode de conversie in vector
Object[] toArray();
Object[] toArray(Object a[]);
}

```

Set modelează noțiunea de *mulțime* în sens matematic. O mulțime nu poate avea elemente duplicate, mai bine zis nu poate conține două obiecte *o1* și *o2* cu proprietatea *o1.equals(o2)*. Moștenește metodele din **Collection**, fără a avea alte metode specifice.

Două dintre clasele standard care oferă implementări concrete ale acestei interfețe sunt **HashSet** și **TreeSet**.

SortedSet este asemănătoare cu interfața **Set**, diferența principală constând în faptul că elementele dintr-o astfel de colecție sunt ordonate ascendent. Pune la dispoziție operații care beneficiază de avantajul ordonării elementelor. Ordonarea elementelor se face conform ordinii lor naturale, sau conform cu ordinea dată de un comparator specificat la crearea colecției și este menținută automat la orice operație efectuată asupra mulțimii. Singura condiție este ca, pentru orice două obiecte *o1, o2* ale colecției, apelul *o1.compareTo(o2)* (sau *comparator.compare(o1, o2)*, dacă este folosit un comparator) trebuie să fie valid și să nu provoace excepții.

Fiind subclasă a interfeței **Set**, moștenește metodele acesteia, oferind metode suplimentare ce țin cont de faptul că mulțimea este sortată:

```

public interface SortedSet extends Set {

    // Subliste
    SortedSet subSet(Object fromElement, Object toElement);
    SortedSet headSet(Object toElement);
}

```

```

SortedSet tailSet(Object fromElement);

// Capete
Object first();
Object last();

Comparator comparator();
}

```

Clasa care implementează această interfață este **TreeSet**.

List descrie *liste (secvențe)* de elemente indexate. Listele pot conține duplicate și permit un control precis asupra poziției unui element prin intermediul indexului acelui element. În plus, față de metodele definite de interfața **Collection**, avem metode pentru acces pozițional, căutare și iterare avansată. Definiția interfeței este:

```

public interface List extends Collection {

    // Acces pozițional
    Object get(int index);
    Object set(int index, Object element);
    void add(int index, Object element);
    Object remove(int index);
    abstract boolean addAll(int index, Collection c);

    // Cautare
    int indexOf(Object o);
    int lastIndexOf(Object o);

    // Iterare
    ListIterator listIterator();
    ListIterator listIterator(int index);

    // Extragere sublista
    List subList(int from, int to);
}

```

Clase standard care implementează această interfață sunt: `ArrayList`, `LinkedList`, `Vector`.

Map descrie structuri de date ce asociază fiecărui element o cheie unică, după care poate fi regăsit. Obiectele de acest tip nu pot conține chei duplicate și fiecare cheie este asociată la un singur element. Ierarhia interfețelor derivate din `Map` este independentă de ierarhia derivată din `Collection`. Definiția interfeței este prezentată mai jos:

```
public interface Map {
    // Metode cu caracter general
    int size();
    boolean isEmpty();
    void clear();

    // Operatii la nivel de element
    Object put(Object key, Object value);
    Object get(Object key);
    Object remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);

    // Operatii la nivel de multime
    void putAll(Map t);

    // Vizualizari ale colectiei
    public Set keySet();
    public Collection values();
    public Set entrySet();

    // Interfata pentru manipularea unei inregistrari
    public interface Entry {
        Object getKey();
        Object getValue();
        Object setValue(Object value);
    }
}
```

Clase care implementează interfața `Map` sunt **HashMap**, **TreeMap** și **Hashtable**.

SortedMap este asemănătoare cu interfața `Map`, la care se adaugă faptul că mulțimea cheilor dintr-o astfel de colecție este menținută ordonată ascendent conform ordinii naturale, sau conform cu ordinea dată de un comparator specificat la crearea colecției. Este subclasa a interfeței `Map`, oferind metode suplimentare pentru: extragere de subtabele, aflarea primei/ultimei chei, aflarea comparatorului folosit pentru ordonare.

Definiția interfeței este dată mai jos:

```
public interface SortedMap extends Map {  
  
    // Extragerea de subtabele  
    SortedMap subMap(Object fromKey, Object toKey);  
    SortedMap headMap(Object toKey);  
    SortedMap tailMap(Object fromKey);  
  
    // Capete  
    Object first();  
    Object last();  
  
    // Comparatorul folosit pentru ordonare  
    Comparator comparator();  
}
```

Clasa care implementează această interfață este **TreeMap**.

7.3 Implementări ale colecțiilor

Înainte de versiunea 1.2, exista un set de clase pentru lucrul cu colecții, însă acestea nu erau organizate pe ierarhia de interfețe prezentată în secțiunea anterioară. Aceste clase sunt în continuare disponibile și multe dintre ele au fost adaptate în așa fel încât să se integreze în noua abordare. Pe lângă acestea au fost create noi clase corespunzătoare interfețelor definite, chiar dacă funcționalitatea lor era aproape identică cu cea a unei clase anterioare.

Clasele de bază care implementează interfețe ce descriu colecții au numele de forma *< Implementare >< Interfața >*, unde 'implementare' se referă la structura internă folosită pentru reprezentarea mulțimii, și sunt prezentate în tabelul de mai jos, împreună cu interfețele corespunzătoare (clasele din vechiul model sunt trecute pe rândul de jos):

Interfața	Clasa
Set	HashSet
SortedSet	TreeSet
List	ArrayList, LinkedList Vector
Map	HashMap Hashtable
SortedMap	TreeMap

Așadar se observă existența unor clase care oferă aceeași funcționalitate, cum ar fi `ArrayList` și `Vector`, `HashMap` și `Hashtable`.

Pe lângă organizarea ierarhică a interfețelor implementate, clasele ce descriu colecții sunt de asemenea concepute într-o manieră ierarhică, ca în figura de mai jos:

```

AbstractCollection - AbstractSet, AbstractList - HashSet,
TreeSet... Vector-Stack
AbstractMap - HashMap, TreeMap, Hashtable
In vechea ierarhie:
Dictionary - Hashtable - Properties

```

Evident, implementarea interfețelor este explicit realizată la nivelul superclaselor abstracte, acestea oferind de altfel și implementări concrete pentru multe din metodele definite de interfețe.

În general, clasele care descriu colecții au unele trăsături comune, cum ar fi:

- permit elementul `null`,
- sunt serializabile,
- au definită metoda `clone`,

- au definită metoda `toString`, care returnează o reprezentare ca șir de caractere a colecției respective,
- permit crearea de iteratori pentru parcurgere,
- au atât constructor fără argumente cât și un constructor care acceptă ca argument o altă colecție
- exceptând clasele din arhitectura veche, nu sunt sincronizate (vezi "Fire de execuție").

7.4 Folosirea eficientă a colecțiilor

După cum am vazut, fiecare interfață ce descrie o colecție are mai multe implementări. De exemplu, interfața `List` este implementată de clasele `ArrayList` și `LinkedList`, prima fiind în general mult mai folosită. De ce există atunci și clasa `LinkedList`? Raspunsul constă în faptul că folosind reprezentări diferite ale mulțimii gestionate putem obține performante mai bune în funcție de situație, prin realizarea unor compromisuri între spațiul necesar pentru memorarea datelor, rapiditatea regăsirii acestora și timpul necesar actualizării colecției în cazul unor modificări.

Să considerăm un exemplu ce creează o listă folosind `ArrayList`, respectiv `LinkedList` și execută diverse operații pe ea, cronometrând timpul necesar realizării acestora:

Listing 7.1: Comparare `ArrayList` - `LinkedList`

```
import java.util.*;

public class TestEficienta {

    final static int N = 100000;

    public static void testAdd(List lst) {
        long t1 = System.currentTimeMillis();
        for(int i=0; i < N; i++)
            lst.add(new Integer(i));
        long t2 = System.currentTimeMillis();
        System.out.println("Add: " + (t2 - t1));
    }
}
```

```

public static void testGet(List lst) {
    long t1 = System.currentTimeMillis();
    for(int i=0; i < N; i++)
        lst.get(i);
    long t2 = System.currentTimeMillis();
    System.out.println("Get: " + (t2 - t1));
}

public static void testRemove(List lst) {
    long t1 = System.currentTimeMillis();
    for(int i=0; i < N; i++)
        lst.remove(0);
    long t2 = System.currentTimeMillis();
    System.out.println("Remove: " + (t2 - t1));
}

public static void main(String args[]) {
    System.out.println("ArrayList");
    List lst1 = new ArrayList();
    testAdd(lst1);
    testGet(lst1);
    testRemove(lst1);

    System.out.println("LinkedList");
    List lst2 = new LinkedList();
    testAdd(lst2);
    testGet(lst2);
    testRemove(lst2);
}
}

```

Timpii aproximativi de rulare pe un calculator cu performanțe medii, exprimați în secunde, sunt dați în tabelul de mai jos:

	ArrayList	LinkedList
add	0.12	0.14
get	0.01	87.45
remove	12.05	0.01

Așadar, adăugarea elementelor este rapidă pentru ambele tipuri de liste. **ArrayList** oferă acces în timp constant la elementele sale și din acest motiv folosirea lui "get" este rapidă, în timp ce pentru **LinkedList** este extrem de lentă, deoarece într-o listă înlănțuită accesul la un element se face prin

parcurgerea secvențială a listei până la elementul respectiv.

La operațiunea de eliminare, folosirea lui `ArrayList` este lentă deoarece elementele rămase suferă un proces de reindexare (shift la stânga), în timp ce pentru `LinkedList` este rapidă și se face prin simpla schimbare a unei legături. Deci, `ArrayList` se comportă bine pentru cazuri în care avem nevoie de regăsirea unor elemente la poziții diferite în listă, iar `LinkedList` funcționează eficient atunci când facem multe operații de modificare (ștergeri, inserări).

Concluzia nu este că una din aceste clase este mai "bună" decât cealaltă, ci că există diferențe substanțiale în reprezentarea și comportamentul diferitelor implementări și că alegerea unei anumite clase pentru reprezentarea unei mulțimi de elemente trebuie să se facă în funcție de natura problemei ce trebuie rezolvată.

7.5 Algoritmi polimorfici

Algoritmii polimorfici descriși în această secțiune sunt metode definite în clasa `Collections` care permit efectuarea unor operații utile cum ar fi căutarea, sortarea, etc. Caracteristicile principale ale acestor algoritmi sunt:

- sunt metode de clasă (statice);
- au un singur argument de tip colecție;
- apelul lor general va fi de forma:
`Collections.algorithm(colectie, [argumente]);`
- majoritatea operează pe liste dar și pe colecții arbitrare.

Metodele mai des folosite din clasa `Collections` sunt:

- **sort** - sortează ascendent o listă referitor la ordinea sa naturală sau la ordinea dată de un comparator;
- **shuffle** - amestecă elementele unei liste - opusul lui **sort**;
- **binarySearch** - efectuează căutarea eficientă (binară) a unui element într-o listă ordonată;

- **reverse** - inversează ordinea elementelor dintr-o listă;
- **fill** - populează o listă cu un anumit element repetat de un număr de ori;
- **copy** - copie elementele unei liste în alta;
- **min** - returnează minimumul dintr-o colecție;
- **max** - returnează maximumul dintr-o colecție;
- **swap** - interschimbă elementele de la două poziții specificate ale unei liste;
- **enumeration** - returnează o enumerare a elementelor dintr-o colecție;
- **unmodifiable** *TipColecție* - returnează o instanță care nu poate fi modificată a colecției respective;
- **synchronized** *TipColecție* - returnează o instanță sincronizată a unei colecții (vezi "Fire de execuție").

7.6 Tipuri generice

Tipurile generice, introduse în versiunea 1.5 a limbajului Java, simplifică lucrul cu colecții, permițând tipizarea elementelor acestora. Definirea unui tip generic se realizează prin specificarea între paranteze unghiulare a unui tip de date Java, efectul fiind impunerea tipului respectiv pentru toate elementele colecției: <**TipDate**>. Să considerăm un exemplu de utilizare a colecțiilor înainte și după introducerea tipurilor generice:

```
// Inainte de 1.5
ArrayList list = new ArrayList();
list.add(new Integer(123));
int val = ((Integer)list.get(0)).intValue();
```

În exemplul de mai sus, lista definită poate conține obiecte de orice tip, deși am dori ca elementele să fie doar numere întregi. Mai mult, trebuie să facem cast explicit de la tipul `Object` la `Integer` atunci când preluăm valoarea unui element.

Folosind tipuri generice, putem rescrie secvența astfel:

```
// Dupa 1.5, folosind tipuri generice
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(new Integer(123));
int val = list.get(0).intValue();
```

Dacă utilizăm și mecanismul de *autoboxing*, obținem o variantă mult simplificată a secvenței inițiale:

```
// Dupa 1.5, folosind si autoboxing
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(123);
int val = list.get(0);
```

În cazul folosirii tipurilor generice, încercarea de a utiliza în cadrul unei colecții a unui element necorespunzător ca tip va produce o eroare la compilare, spre deosebire de varianta anterioară ce permitea doar aruncarea unor excepție de tipul `ClassCastException` în cazul folosirii incorecte a tipurilor.

7.7 Iteratori și enumerări

Enumerările și iteratorii descriu modalități pentru parcurgerea secvențială a unei colecții, indiferent dacă aceasta este indexată sau nu. Ei sunt descriși de obiecte ce implementează interfețele **Enumeration**, respectiv **Iterator** sau **ListIterator**. Toate clasele care implementează colecții au metode ce returnează o enumerare sau un iterator pentru parcurgerea elementelor lor. Deoarece funcționalitatea interfeței **Enumeration** se regăsește în **Iterator**, aceasta din urmă este preferată în noile implementări ale colecțiilor.

Metodele uzuale ale acestor interfețe sunt prezentate mai jos, împreună cu modalitatea lor de folosire, semnificațiile lor fiind evidente:

- **Enumeration**: `hasMoreElements`, `nextElement`

```
// Parcurgerea elementelor unui vector v
Enumeration e = v.elements();
while (e.hasMoreElements()) {
    System.out.println(e.nextElement());
}
// sau, varianta mai concisa
for (Enumeration e = v.elements());
```

```

        e.hasMoreElements();) {
    System.out.println(e.nextElement());
}

```

- **Iterator**: hasNext, next, remove

```

// Parcurgerea elementelor unui vector
// si eliminarea elementelor nule
for (Iterator it = v.iterator(); it.hasNext();) {
    Object obj = it.next();
    if (obj == null)
        it.remove();
}

```

- **ListIterator**: hasNext, hasPrevious, next, previous, remove, add, set

```

// Parcurgerea elementelor unui vector
// si inlocuirea elementelor nule cu 0
for (ListIterator it = v.listIterator();
    it.hasNext();) {
    Object obj = it.next();
    if (obj == null)
        it.set(new Integer(0));
}

```

Iteratorii simpli permit eliminarea elementului curent din colecția pe care o parcurg, cei de tip `ListIterator` permit și inserarea unui element la poziția curentă, respectiv modificarea elementului curent, precum și iterarea în ambele sensuri. Iteratorii sunt preferați enumerărilor datorită posibilității lor de a acționa asupra colecției pe care o parcurg prin metode de tip `remove`, `add`, `set` dar și prin faptul că denumirile metodelor sunt mai concise.

Atenție

Deoarece colecțiile sunt construite peste tipul de date `Object`, metodele de tip `next` sau `prev` ale iteratorilor vor returna tipul `Object`, fiind responsabilitatea noastră de a face conversie (cast) la alte tipuri de date, dacă este cazul.

În exemplul de mai jos punem într-un vector numerele de la 1 la 10, le amestecăm, după care le parcurgem element cu element folosind un iterator, înlocuind numerele pare cu 0.

Listing 7.2: Folosirea unui iterator

```
import java.util.*;
class TestIterator {
    public static void main(String args[]) {
        ArrayList a = new ArrayList();

        // Adaugam numerele de la 1 la 10
        for(int i=1; i<=10; i++)
            a.add(new Integer(i));

        // Amestecam elementele colectiei
        Collections.shuffle(a);
        System.out.println("Vectorul amestecat: " + a);

        // Parcurgem vectorul
        for(ListIterator it=a.listIterator(); it.hasNext(); ) {
            Integer x = (Integer) it.next();

            // Daca elementul curent este par, il facem 0
            if (x.intValue() % 2 == 0)
                it.set(new Integer(0));
        }
        System.out.print("Rezultat: " + a);
    }
}
```

Începând cu versiunea 1.5 a limbajului Java, există o variantă simplificată de utilizare a iteratorilor. Astfel, o secvență de genul:

```
ArrayList<Integer> list = new ArrayList<Integer>();
for (Iterator i = list.iterator(); i.hasNext();) {
    Integer val=(Integer)i.next();
    // Proceseaza val
    ...
}
```

```
}
```

poate fi rescrisă astfel:

```
ArrayList<Integer> list = new ArrayList<Integer>();  
for (Integer val : list) {  
    // Proceseaza val  
    ...  
}
```


Capitolul 8

Serializarea obiectelor

8.1 Folosirea serializării

Definiție

Serializarea este o metodă ce permite transformarea unui obiect într-o secvență de octeți sau caractere din care să poată fi refăcut ulterior obiectul original.

Cu alte cuvinte, serializarea permite salvarea într-o manieră unitară a tuturor informațiilor unui obiect pe un mediu de stocare extern programului. Procesul invers, de citire a unui obiect serializat pentru a-i reface starea originală, se numește *deserializare*. Într-un cadru mai larg, prin serializare se înțelege procesul de scriere/citire a obiectelor.

Tipurile primitive pot fi de asemenea serializate.

Utilitatea serializării constă în următoarele aspecte:

- Asigură un mecanism *simplu de utilizat* pentru salvarea și restaurarea a datelor.
- Permite *persistența obiectelor*, ceea ce înseamnă că durata de viața a unui obiect nu este determinată de execuția unui program în care acesta este definit - obiectul poate exista și între apelurile programelor care îl folosesc. Acest lucru se realizează prin serializarea obiectului și scrierea lui pe disc înainte de terminarea unui program, apoi, la relansarea programului, obiectul va fi citit de pe disc și starea lui refăcută. Acest

tip de persistență a obiectelor se numește *persistență ușoară*, întrucât ea trebuie efectuată explicit de către programator și nu este realizată automat de către sistem.

- *Compensarea diferențelor între sisteme de operare* - transmiterea unor informații între platforme de lucru diferite se realizează unitar, independent de formatul de reprezentare a datelor, ordinea octeților sau alte detalii specifice sistemelor repective.
- *Transmiterea datelor în rețea* - Aplicațiile ce rulează în rețea pot comunica între ele folosind fluxuri pe care sunt trimise, respectiv recepționate obiecte serializate.
- *RMI (Remote Method Invocation)* - este o modalitate prin care metodele unor obiecte de pe o altă mașină pot fi apelate ca și cum acestea ar exista local pe mașina pe care rulează aplicația. Atunci când este trimis un mesaj către un obiect "remote" (de pe altă mașină), serializarea este utilizată pentru transportul argumentelor prin rețea și pentru returnarea valorilor.
- *Java Beans* - sunt componente reutilizabile, de sine stătătoare ce pot fi utilizate în medii vizuale de dezvoltare a aplicațiilor. Orice componentă Bean are o stare definită de valorile implicite ale proprietăților sale, stare care este specificată în etapa de design a aplicației. Mediile vizuale folosesc mecanismul serializării pentru asigurarea persistenței componentelor Bean.

Un aspect important al serializării este că nu salvează doar imaginea unui obiect ci și toate referințele la alte obiecte pe care acesta le conține. Acesta este un proces recursiv de salvare a datelor, întrucât celelalte obiectele referite de obiectul care se serializează pot referi la rândul lor alte obiecte, și așa mai departe. Așadar referințele care construiesc starea unui obiect formează o întregă rețea, ceea ce înseamnă că un algoritm general de salvare a stării unui obiect nu este tocmai facil.

În cazul în care starea unui obiect este formată doar din valori ale unor variabile de tip primitiv, atunci salvarea informațiilor încapsulate în acel obiect se poate face și prin salvarea pe rând a datelor, folosind clasa `DataOutputStream`, pentru ca apoi să fie restaurate prin metode ale clasei `DataInputStream`, dar, așa cum am văzut, o asemenea abordare nu este

în general suficientă, deoarece pot apărea probleme cum ar fi: variabilele membre ale obiectului pot fi instanțe ale altor obiecte, unele câmpuri pot face referință la același obiect, etc.

Serializarea în format binar a tipurilor primitive și a obiectelor se realizează prin intermediul fluxurilor definite de clase specializate în acest scop cu ar fi:

`ObjectOutputStream` pentru scriere și `ObjectInputStream` pentru restaurare.

În continuare, prin termenul *serializare* ne vom referi doar la serializarea în format binar.

8.1.1 Serializarea tipurilor primitive

Serializarea tipurilor primitive poate fi realizată fie prin intermediul fluxurilor `DataOutputStream` și `DataInputStream`, fie cu `ObjectOutputStream` și `ObjectInputStream`. Acestea implementează interfețele `DataInput`, respectiv `DataOutput` ce declară metode de tipul `readTipPrimitiv`, respectiv `writeTipPrimitiv` pentru scrierea/citirea datelor primitive și a șirurilor de caractere.

Mai jos este prezentat un exemplu de serializare folosind clasa `DataOutputStream`:

```
FileOutputStream fos = new FileOutputStream("test.dat");
DataOutputStream out = new DataOutputStream(fos);
out.writeInt(12345);
out.writeDouble(12.345);
out.writeBoolean(true);
out.writeUTF("Sir de caractere");
out.flush();
fos.close();
```

Citirea informațiilor scrise în exemplul de mai sus se va face astfel:

```
FileInputStream fis = new FileInputStream("test.dat");
DataInputStream in = new DataInputStream(fis);
int i = in.readInt();
double d = in.readDouble();
boolean b = in.readBoolean();
```

```
String s = in.readUTF();  
fis.close();
```

8.1.2 Serializarea obiectelor

Serializarea obiectelor se realizează prin intermediul fluxurilor definite de clasele **ObjectOutputStream** (pentru salvare) și **ObjectInputStream** (pentru restaurare). Acestea sunt fluxuri de procesare, ceea ce înseamnă că vor fi folosite împreună cu alte fluxuri pentru scrierea/citirea efectivă a datelor pe mediul extern pe care va fi salvat, sau de pe care va fi restaurat un obiect serializat.

Mecanismul implicit de serializare a unui obiect va salva numele clasei obiectului, signatura clasei și valorile tuturor câmpurile serializabile ale obiectului. Referințele la alte obiecte serializabile din cadrul obiectului curent vor duce automat la serializarea acestora iar referințele multiple către un același obiect sunt codificate utilizând un algoritm care să poată reface ”rețeaua de obiecte” la aceeași stare ca atunci când obiectul original a fost salvat.

Clasele **ObjectInputStream** și **ObjectOutputStream** implementează interfețele **ObjectInput**, respectiv **ObjectOutput** care extind **DataInput**, respectiv **DataOutput**, ceea ce înseamnă că, pe lângă metodele dedicate serializării obiectelor, vor exista și metode pentru scrierea/citirea datelor primitive și a șirurilor de caractere.

Metodele pentru serializarea obiectelor sunt:

- **writeObject**, pentru scriere și
- **readObject**, pentru restaurare.

8.1.3 Clasa **ObjectOutputStream**

Scrierea obiectelor pe un flux de ieșire este un proces extrem de simplu, secvența uzuală fiind cea de mai jos:

```
ObjectOutputStream out = new ObjectOutputStream(fluxPrimitiv);  
out.writeObject(referintaObiect);  
out.flush();  
fluxPrimitiv.close();
```

Exemplul de mai jos construiește un obiect de tip `Date` și îl salvează în fișierul `test.ser`, împreună cu un obiect de tip `String`. Evident, fișierul rezultat va conține informațiile reprezentate în format binar.

```
FileOutputStream fos = new FileOutputStream("test.ser");
ObjectOutputStream out = new ObjectOutputStream(fos);
out.writeObject("Ora curenta:");
out.writeObject(new Date());
out.flush();
fos.close();
```

Deoarece implementează interfața `DataOutput`, pe lângă metoda de scriere a obiectelor, clasa pune la dispoziție și metode de tipul `writeTipPrimitiv` pentru serializarea tipurilor de date primitive și a șirurilor de caractere, astfel încât apeluri ca cele de mai jos sunt permise :

```
out.writeInt(12345);
out.writeDouble(12.345);
out.writeBoolean(true);
out.writeUTF("Sir de caractere");
```

Metoda `writeObject` aruncă excepții de tipul `IOException` și derivate din aceasta, mai precis `NotSerializableException` dacă obiectul primit ca argument nu este serializabil, sau `InvalidClassException` dacă sunt probleme cu o clasă necesară în procesul de serializare. Vom vedea în continuare că un obiect este serializabil dacă este instanță a unei clase ce implementează interfața `Serializable`.

8.1.4 Clasa `ObjectInputStream`

Odată ce au fost scrise obiecte și tipuri primitive de date pe un flux, citirea acestora și reconstruirea obiectelor salvate se va face printr-un flux de intrare de tip `ObjectInputStream`. Acesta este de asemenea un flux de procesare și va trebui asociat cu un flux pentru citirea efectivă a datelor, cum ar fi `FileInputStream` pentru date salvate într-un fișier. Secvența uzuală pentru deserializare este cea de mai jos:

```
ObjectInputStream in = new ObjectInputStream(fluxPrimitiv);
Object obj = in.readObject();
//sau
```

```
TipReferinta ref = (TipReferinta)in.readObject();
fluxPrimitiv.close();
```

Citirea informațiilor scrise în exemplul de mai sus se va face astfel:

```
FileInputStream fis = new FileInputStream("test.ser");
ObjectInputStream in = new ObjectInputStream(fis);
String mesaj = (String)in.readObject();
Date data = (Date)in.readObject();
fis.close();
```

Trebuie observat că metoda `readObject` are tipul returnat `Object`, ceea ce înseamnă că trebuie realizată explicit conversia la tipul corespunzător obiectului citit:

```
Date date = in.readObject(); // gresit
Date date = (Date)in.readObject(); // corect
```

Atenție

Ca și la celelalte fluxuri de date care implementează interfața `DataInput` citirea dintr-un flux de obiecte trebuie să se facă exact în ordinea în care acestea au fost scrise, altfel vor apărea evident excepții în procesul de deserializare.

Clasa `ObjectInputStream` implementează interfața `DataInput` deci, pe lângă metoda de citire a obiectelor, clasa pune la dispoziție și metode de tipul `readTipPrimitiv` pentru citirea tipurilor de date primitive și a șirurilor de caractere.

```
int i = in.readInt();
double d = in.readDouble();
boolean b = in.readBoolean();
String s = in.readUTF();
```

8.2 Obiecte serializabile

Un obiect este serializabil dacă și numai dacă clasa din care face parte implementează interfața **Serializable**. Așadar, dacă dorim ca instanțele unei clase să poată fi serializate, clasa respectivă trebuie să implementeze, direct sau indirect, interfața **Serializable**.

8.2.1 Implementarea interfeței **Serializable**

Interfața **Serializable** nu conține nici o declarație de metodă sau constantă, singurul ei scop fiind de a identifica clasele ale căror obiecte sunt serializabile. Definiția sa completă este:

```
package java.io;
public interface Serializable {
    // Nimic !
}
```

Declararea claselor ale căror instanțe trebuie să fie serializate este așadar extrem de simplă, fiind făcută prin simpla implementare a interfeței **Serializable**:

```
public class ClasaSerializabila implements Serializable {
    // Corpul clasei
}
```

Orice subclasă a unei clase serializabile este la rândul ei serializabilă, întrucât implementează indirect interfața **Serializable**.

În situația în care dorim să declarăm o clasă serializabilă dar superclasa sa nu este serializabilă, atunci trebuie să avem în vedere următoarele lucruri:

- Variabilele accesibile ale superclasei nu vor fi serializate, fiind responsabilitatea clasei curente de a asigura un mecanism propriu pentru salvarea/restaurarea lor. Acest lucru va fi discutat în secțiunea referitoare la personalizarea serializării.
- Superclasa trebuie să aibă obligatoriu un constructor accesibil fără argumente, acesta fiind utilizat pentru inițializarea variabilelor moștenite în procesul de restaurare al unui obiect. Variabilele proprii vor fi inițializate cu valorile de pe fluxul de intrare. În lipsa unui constructor accesibil fără argumente pentru superclasă, va fi generată o excepție la execuție.

În procesul serializării, dacă este întâlnit un obiect care nu implementează interfața `Serializable` atunci va fi generată o excepție de tipul `NotSerializableException` ce va identifica respectiva clasă neserializabilă.

8.2.2 Controlul serializării

Există cazuri când dorim ca unele variabile membre ale unui obiect să nu fie salvate automat în procesul de serializare. Acestea sunt cazuri comune atunci când respectivele câmpuri reprezintă informații confidențiale, cum ar fi parole, sau variabile temporare pe care nu are rost să le salvăm. Chiar declarate private în cadrul clasei aceste câmpuri participă la serializare. Pentru ca un câmp să nu fie salvat în procesul de serializare el trebuie declarat cu modificatorul **transient** și trebuie să fie ne-static. De exemplu, declararea unei variabile membre temporare ar trebui făcută astfel:

```
transient private double temp;  
// Ignorata la serializare
```

Modificatorul `static` anulează efectul modificatorului `transient`. Cu alte cuvinte, variabilele de clasă participă obligatoriu la serializare.

```
static transient int N;  
// Participa la serializare
```

În exemplele următoare câmpurile marcate 'DA' participă la serializare, cele marcate 'NU', nu participă iar cele marcate cu 'Excepție' vor provoca excepții de tipul `NotSerializableException`.

Listing 8.1: Modificatorii `static` și `transient`

```
import java.io.*;  
  
public class Test1 implements Serializable {  
  
    int x=1;                //DA  
    transient int y=2;      //NU  
    transient static int z=3; //DA  
    static int t=4;         //DA  
  
    public String toString() {  
        return x + ", " + y + ", " + z + ", " + t;  
    }  
}
```

Dacă un obiect ce trebuie serializat are referințe la obiecte neserializabile, atunci va fi generată o excepție de tipul `NotSerializableException`.

Listing 8.2: Membrii neserializabili

```
import java.io.*;

class A {
    int x=1;
}

class B implements Serializable {
    int y=2;
}

public class Test2 implements Serializable{
    A a = new A(); //Excepție
    B b = new B(); //DA

    public String toString() {
        return a.x + ", " + b.y;
    }
}
```

Atunci când o clasă serializabilă deriva dintr-o altă clasă, salvarea câmpurilor clasei părinte se va face doar dacă și aceasta este serializabilă. În caz contrar, subclasa trebuie să salveze explicit și câmpurile moștenite.

Listing 8.3: Serializarea câmpurilor moștenite

```
import java.io.*;
class C {
    int x=0;
    // Obligativ constructor fara argumente
}

class D extends C implements Serializable {
    int y=0;
}

public class Test3 extends D {
    public Test3() {
        x = 1; //NU
        y = 2; //DA
    }
}
```

```
    }  
    public String toString() {  
        return x + ", " + y;  
    }  
}
```

Mai jos este descrisa o aplicație care efectuează salvarea și restaurarea unor obiecte din cele trei clase prezentate mai sus.

Listing 8.4: Testarea serializării

```
import java.io.*;  
  
public class Exemplu {  
  
    public static void test(Object obj) throws IOException {  
        // Salvam  
        FileOutputStream fos = new FileOutputStream("fisier.ser")  
            ;  
        ObjectOutputStream out = new ObjectOutputStream(fos);  
  
        out.writeObject(obj);  
        out.flush();  
  
        fos.close();  
        System.out.println("A fost salvat obiectul: " + obj);  
  
        // Restauram  
        FileInputStream fis = new FileInputStream("fisier.ser");  
        ObjectInputStream in = new ObjectInputStream(fis);  
        try {  
            obj = in.readObject();  
        } catch(ClassNotFoundException e) {  
            e.printStackTrace();  
        }  
  
        fis.close();  
        System.out.println("A fost restaurat obiectul: " + obj);  
    }  
  
    public static void main(String args[]) throws IOException {  
        test(new Test1());  
        try {  
            test(new Test2());  
        } catch(NotSerializableException e) {
```

```
        System.out.println("Obiect neserializabil: " + e);
    }

    test(new Test3());
}
}
```

Rezultatul acestui program va fi :

```
A fost salvat obiectul: 1, 2, 3, 4
A fost restaurat obiectul: 1, 0, 3, 4
Obiect neserializabil: java.io.NotSerializableException: A
A fost salvat obiectul: 1, 2
A fost restaurat obiectul: 0, 2
```

8.3 Personalizarea serializării obiectelor

Dezavantajul mecanismului implicit de serializare este că algoritmul pe care se beazează, fiind creat pentru cazul general, se poate comporta ineficient în anumite situații: poate fi mult mai lent decât este cazul sau reprezentarea binară generată pentru un obiect poate fi mult mai voluminoasă decât ar trebui. În aceste situații, putem să înlocuim algoritmul implicit cu unul propriu, particularizat pentru o clasă anume. De asemenea, este posibil să extindem comportamentul implicit, adăugând și alte informații necesare pentru serializarea unor obiecte.

În majoritatea cazurilor mecanismul standard este suficient însă, după cum am spus, o clasă poate avea nevoie de mai mult control asupra serializării.

Personalizarea serializării se realizează prin definirea (într-o clasă serializabilă!) a metodelor `writeObject` și `readObject` având **exact** semnătura de mai jos:

```
private void writeObject(java.io.ObjectOutputStream stream)
    throws IOException
private void readObject(java.io.ObjectInputStream stream)
    throws IOException, ClassNotFoundException
```

Metoda `writeObject` controlează ce date sunt salvate iar `readObject` controlează modul în care sunt restaurate obiectele, citind informațiile salvate și, eventual, modificând starea obiectelor citite astfel încât ele să corespundă

anumitor cerințe. În cazul în care nu dorim să înlocuim complet mecanismul standard, putem să folosim metodele **defaultWriteObject**, respectiv **defaultReadObject** care descriu procedurile implicite de serializare.

Forma generală de implementare a metodelor `writeObject` și `readObject` este:

```
private void writeObject(ObjectOutputStream stream)
    throws IOException {

    // Procesarea campurilor clasei (criptare, etc.)
    ...
    // Scrierea obiectului curent
    stream.defaultWriteObject();

    // Adaugarea altor informatii suplimentare
    ...
}

private void readObject(ObjectInputStream stream)
    throws IOException, ClassNotFoundException {

    // Restaurarea obiectului curent
    stream.defaultReadObject();

    // Actualizarea starii obiectului (decriptare, etc.)
    // si extragerea informatiilor suplimentare
    ...
}
```

Metodele `writeObject` și `readObject` sunt responsabile cu serializarea clasei în care sunt definite, serializarea superclasei sale fiind făcută automat (și implicit). Dacă însă o clasă trebuie să-și coordoneze serializarea proprie cu serializarea superclasei sale, atunci trebuie să implementeze interfața `Externalizable`.

8.3.1 Controlul versiunilor claselor

Să presupunem că dorim să realizăm o aplicație care să țină evidența angajaților unei companii. Evident, vom avea nevoie de o clasă care să reprezinte

noțiunea de *angajat*. O variantă simplificată a acesteia ar putea arăta astfel:

Listing 8.5: Prima variantă a clasei Angajat

```
import java.io.*;

class Angajat implements Serializable {

    public String nume;
    public int salariu;
    private String parola;

    public Angajat(String nume, int salariu, String parola) {
        this.nume = nume;
        this.salariu = salariu;
        this.parola = parola;
    }

    public String toString() {
        return nume + " (" + salariu + ")";
    }

}
```

Mai jos este prezentată o mică aplicație care permite introducerea de angajați și salvarea lor într-un fișier. La fiecare pornire a aplicației, vor fi citite datele din fișier astfel încât programul va actualiza în permanentă lista angajaților cu noi persoane.

Listing 8.6: Aplicația de gestionare a angajaților

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

public class GestiuneAngajati {

    //Lista angajatilor
    ArrayList ang = new ArrayList();

    public void citire() throws IOException {
        FileInputStream fis = null;
        try {
            fis = new FileInputStream("angajati.ser");
            ObjectInputStream in = new ObjectInputStream(fis);
```

```
        ang = (ArrayList) in.readObject();
    } catch(FileNotFoundException e) {
        System.out.println("Fisierul nou...");
    } catch(Exception e) {
        System.out.println("Eroare la citirea datelor...");
        e.printStackTrace();
    }finally {
        if (fis != null)
            fis.close();
    }
    System.out.println("Lista angajatilor:\n" + ang);
}

public void salvare() throws IOException {
    FileOutputStream fos =
        new FileOutputStream("angajati.ser");
    ObjectOutputStream out = new ObjectOutputStream(fos);
    out.writeObject(ang);
}

public void adaugare() throws IOException {
    BufferedReader stdin = new BufferedReader(
        new InputStreamReader(System.in));

    while (true) {
        System.out.print("\nNume:");
        String nume = stdin.readLine();

        System.out.print("Salariu:");
        int salariu = Integer.parseInt(stdin.readLine());

        System.out.print("Parola:");
        String parola = stdin.readLine();

        ang.add(new Angajat(nume, salariu, parola));

        System.out.print("Mai adaugati ? (D/N)");
        String raspuns = stdin.readLine().toUpperCase();
        if (raspuns.startsWith("N"))
            break;
    }
}

public static void main(String args[]) throws IOException {
    GestiuneAngajati app = new GestiuneAngajati();
```

```
//Incarcam angajatii din fisier
app.citire();

//Adaugam noi angajati
app.adaugare();

//Salvam angajatii inapoi fisier
app.salvare();
}
}
```

Problema care se pune acum este următoarea. După introducerea unui număr suficient de mare de angajați în fișier, clasa `Angajat` este modificată prin adăugarea unei noi variabile membre care să rețină *adresa*. La execuția aplicației noastre, procedura de citire a angajaților din fișier nu va mai funcționa, producând o excepție de tipul `InvalidClassException`. Această problemă ar fi apărut chiar dacă variabila adăugată era declarată de tip `transient`. De ce se întâmplă acest lucru ?

Explicația constă în faptul că mecanismul de serializare Java este foarte atent cu semnătura claselor serializate. Pentru fiecare obiect serializat este calculat automat un număr reprezentat pe 64 de biți, care reprezintă un fel de "amprentă" a clasei obiectului. Acest număr, denumit `serialVersionUID`, este generat pornind de la diverse informații ale clasei, cum ar fi variabilele sale membre, (dar nu numai) și este salvat în procesul de serializare împreună cu celelalte date. În plus, orice modificare semnificativă a clasei, cum ar fi adăugarea unui nou câmp, va determina modificarea numărului său de versiune.

La restaurarea unui obiect, numărul de versiune salvat în forma serializată va fi regăsit și comparat cu noua semnătură a clasei obiectului. În cazul în care acestea nu sunt egale, va fi generată o excepție de tipul `InvalidClassException` și deserializarea nu va fi făcută.

Această abordare extrem de precaută este foarte utilă pentru prevenirea unor anomalii ce pot apărea când două versiuni de clase sunt incompatibile, dar poate fi supărătoare atunci când modificările aduse clasei nu strică compatibilitatea cu vechea versiune. În această situație trebuie să comunicăm explicit că cele două clase sunt compatibile. Acest lucru se realizează prin setarea manuală a variabilei `serialVersionUID` în cadrul clasei dorite, adăugând pur și simplu câmpul:


```
static final long serialVersionUID = /* numar_serial_clasa */;
```

Prezența variabilei `serialVersionUID` printre membrii unei clase va informa algoritmul de serializare că nu mai calculeze numărul de serie al clasei, ci să-l folosească pe cel specificat de noi. Cum putem afla însă numărul de serie al vechii clase `Angajat` care a fost folosită anterior la salvarea angajaților?

Utilitarul `serialVer` permite generarea numărului `serialVersionUID` pentru o clasă specificată. Așadar, trebuie să recompilăm vechea clasă `Angajat` și să-i aflăm numărul de serie astfel: `serialVer Angajat`. Rezultatul va fi:

`Angajat:`

```
static final long serialVersionUID = 5653493248680665297L;
```

Vom rescrie noua clasă `Angajat` astfel încât să fie compatibilă cu cea veche astfel:

Listing 8.7: Variantă compatibilă a clasei `Angajat`

```
import java.io.*;

class Angajat implements Serializable {

    static final long serialVersionUID = 5653493248680665297L;

    public String nume, adresa;
    public int salariu;
    private String parola;

    public Angajat(String nume, int salariu, String parola) {
        this.nume = nume;
        this.adresa = "Iasi";
        this.salariu = salariu;
        this.parola = parola;
    }

    public String toString() {
        return nume + " (" + salariu + ")";
    }

}
```

Aplicația noastră va funcționa acum, însă rubrica *adresă* nu va fi inițializată în nici un fel (va fi null), deoarece ea nu exista în formatul original. La noua

salvare a datelor, vor fi serializate și informațiile legate de adresă (evident, trebuie însă să le citim de la tastatură...)

8.3.2 Securizarea datelor

După cum am văzut membrii privați, cum ar fi *parola* din exemplul de mai sus, participă la serializare. Problema constă în faptul că, deși în format binar, informațiile unui obiect serializat nu sunt criptate în nici un fel și pot fi regăsite cu ușurință, ceea ce poate reprezenta un inconvenient atunci când există câmpuri confidențiale.

Rezolvarea acestei probleme se face prin modificarea mecanismului implicit de serializare, implementând metodele **readObject** și **writeObject**, precum și prin utilizarea unei funcții de criptare a datelor. Varianta securizată a clasei **Angajat** din exemplul anterior ar putea arăta astfel:

Listing 8.8: Varianta securizată a clasei **Angajat**

```
import java.io.*;

class Angajat implements Serializable {

    static final long serialVersionUID = 5653493248680665297L;

    public String nume, adresa;
    public int salariu;
    private String parola;

    public Angajat(String nume, int salariu, String parola) {
        this.nume = nume;
        this.adresa = "Iasi";
        this.salariu = salariu;
        this.parola = parola;
    }

    public String toString() {
        return nume + " (" + salariu + ")";
    }

    static String criptare(String input, int offset) {
        StringBuffer sb = new StringBuffer();
        for (int n=0; n<input.length(); n++)
            sb.append((char)(offset+input.charAt(n)));
        return sb.toString();
    }
}
```

```
    }

    private void writeObject(ObjectOutputStream stream)
        throws IOException {
        parola = criptare(parola, 3);
        stream.defaultWriteObject();
        parola = criptare(parola, -3);
    }

    private void readObject(ObjectInputStream stream)
        throws IOException, ClassNotFoundException {
        stream.defaultReadObject();
        parola = criptare(parola, -3);
    }
}
```

8.3.3 Implementarea interfeței Externalizable

Pentru un control complet, explicit, al procesului de serializare, o clasă trebuie să implementeze interfața **Externalizable**. Pentru instanțe ale acestor clase doar numele clasei este salvat automat pe fluxul de obiecte, clasa fiind responsabilă cu scrierea și citirea membrilor săi și trebuie să se coordoneze cu superclasele ei.

Definiția interfeței **Externalizable** este:

```
package java.io;
public interface Externalizable extends Serializable {
    public void writeExternal(ObjectOutput out)
        throws IOException;
    public void readExternal(ObjectInput in)
        throws IOException, ClassNotFoundException;
}
```

Așadar, aceste clase trebuie să implementeze obligatoriu metodele **writeExternal** și **readExternal** în care se va face serializarea completă a obiectelor și coordonarea cu superclasa ei.

Uzual, interfața **Externalizable** este folosită în situații în care se dorește îmbunătățirea performanțelor algoritmului standard, mai exact creșterea vitezei procesului de serializare.

Mai jos este prezentată o clasă simplă și modalitatea de rescriere a sa folosind interfața `Externalizable`:

Listing 8.9: Serializare implicită

```
import java.io.*;

class Persoana implements Serializable {
    int cod;
    String nume;

    public Persoana(String nume, int cod) {
        this.nume = nume;
        this.cod = cod;
    }
}
```

Listing 8.10: Serializare proprie

```
import java.io.*;

class Persoana implements Externalizable {
    int cod;
    String nume;

    public Persoana(String nume, int cod) {
        this.nume = nume;
        this.cod = cod;
    }

    public void writeExternal(ObjectOutput s)
        throws IOException {
        s.writeUTF(nume);
        s.writeInt(cod);
    }

    public void readExternal(ObjectInput s)
        throws ClassNotFoundException, IOException {
        nume = s.readUTF();
        cod = s.readInt();
    }
}
```

8.4 Clonarea obiectelor

Se știe că nu putem copia valoarea unui obiect prin instrucțiunea de atribuire. O secvența de forma:

```
TipReferinta o1 = new TipReferinta();
TipReferinta o2 = o1;
```

nu face decât să declare o nouă variabilă *o2* ca referința la obiectul referit de *o1* și nu creează sub nici o formă un nou obiect.

O posibilitate de a face o copie a unui obiect este folosirea metodei **clone** definită în clasa **Object**. Aceasta creează un nou obiect și inițializează toate variabilele sale membre cu valorile obiectului clonat.

```
TipReferinta o1 = new TipReferinta();
TipReferinta o2 = (TipReferinta) o1.clone();
```

Deficiența acestei metode este că nu realizează duplicarea întregii rețele de obiecte corespunzătoare obiectului clonat. În cazul în care există câmpuri referința la alte obiecte, obiectele referite nu vor mai fi clonate la rândul lor.

O metodă **clone** care să realizeze o copie efectivă a unui obiect, împreună cu copierea tuturor obiectelor referite de câmpurile acelui obiect poate fi implementată prin mecanismul serializării astfel:

```
public Object clone() {
    try {
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        ObjectOutputStream out = new ObjectOutputStream(baos);
        out.writeObject(this);
        out.close();

        byte[] buffer = baos.toByteArray();
        ByteArrayInputStream bais = new ByteArrayInputStream(buffer);
        ObjectInputStream in = new ObjectInputStream(bais);
        Object ret = in.readObject();
        in.close();

        return ret;
    } catch (Exception e) {
```

```
        System.out.println(e);  
        return null;  
    }  
}
```


Capitolul 9

Interfața grafică cu utilizatorul

9.1 Introducere

Interfața grafică cu utilizatorul (GUI), este un termen cu înțeles larg care se referă la toate tipurile de comunicare *vizuală* între un program și utilizatorii săi. Aceasta este o particularizare a interfeței cu utilizatorul (UI), prin care vom înțelege conceptul generic de interacțiune dintre program și utilizatori. Limbajul Java pune la dispoziție numeroase clase pentru implementarea diverselor funcționalități UI, însă ne vom ocupa în continuare de cele care permit realizarea interfeței grafice cu utilizatorul (GUI).

De la apariția 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 dificultății legate de implementarea noțiunii de portabilitate, pe de altă parte nevoii de a integra mecanismele GUI cu tehnologii apărute și dezvoltate ulterior, cum ar fi *Java Beans*. În momentul actual, există două modalități de a crea o aplicație cu interfață grafică și anume:

- **AWT** (Abstract Windowing Toolkit) - este API-ul inițial pus la dispoziție 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 și IBM, Swing se bazează pe modelul AWT, extinzând funcționalitatea acestuia și adăugând sau înlocuind componente pentru dezvoltarea aplicațiilor GUI.

Așadar, este de preferat ca aplicațiile Java să fie create folosind tehnologia Swing, aceasta punând la dispoziție o paletă mult mai largă de facilități, însă nu vom renunța complet la AWT deoarece aici există clase esențiale, reutilizate în Swing.

În acest capitol vom prezenta clasele de bază și mecanismul de tratare a evenimentelor din AWT, deoarece va fi simplificat procesul de înțelegere a dezvoltării unei aplicații GUI, după care vom face trecerea la Swing.

În principiu, crearea unei aplicații grafice presupune următoarele lucruri:

- **Design**

- Crearea unei suprafețe de afișare (cum ar fi o fereastră) pe care vor fi așezate obiectele grafice (componente) care servesc la comunicarea cu utilizatorul (butoane, controale pentru editarea textelor, liste, etc);
- Crearea și așezarea componentelor pe suprafața de afișare la pozițiile corespunzătoare;

- **Funcționalitate**

- Definirea unor acțiuni care trebuie să se execute în momentul când utilizatorul interacționează cu obiectele grafice ale aplicației;
- ”Ascultarea” evenimentelor generate de obiecte în momentul interacțiunii cu utilizatorul și executarea acțiunilor corespunzătoare, așa cum au fost ele definite.

9.2 Modelul AWT

Pachetul care oferă componente AWT este **java.awt**.

Obiectele grafice sunt derivate din **Component**, cu excepția meniurilor care descind din clasa **MenuComponent**. Așadar, prin noțiunea de componentă vom înțelege în continuare orice obiect care poate avea o reprezentare grafică și care poate interacționa cu utilizatorul. Exemple de componente sunt ferestrele, butoanele, listele, bare de defilare, etc. Toate componentele AWT sunt definite de clase proprii ce se găsesc în pachetul **java.awt**, clasa **Component** fiind superclasa abstractă a tuturor acestor clase.

Crearea obiectelor grafice nu realizează automat și afișarea lor pe ecran. Mai întâi ele trebuie așezate pe o *suprafața de afișare*, care poate fi o fereastră

sau un applet, și vor deveni vizibile în momentul în care suprafața pe care sunt afișate va fi vizibilă. O astfel de suprafață pe care sunt plasate componente se mai numește *container* și reprezintă o instanță a unei clase derivate din **Container**. Clasa **Container** este o subclasă aparte a lui **Component**, fiind la rândul ei superclasa tuturor suprafețelor de afișare Java.

Așa cum am văzut, interfață grafică servește interacțiunii cu utilizatorul. De cele mai multe ori programul trebuie să facă o anumită prelucrare în momentul în care utilizatorul a efectuat o acțiune și, prin urmare, componentele trebuie să genereze evenimente în funcție de acțiunea pe care au suferit-o (acțiune transmisă de la tastatură, mouse, etc.). Incepând cu versiunea 1.1 a limbajului Java, evenimentele sunt instanțe ale claselor derivate din **AWTEvent**.

Așadar, un *eveniment* este produs de o acțiune a utilizatorului asupra unui obiect grafic, deci evenimentele nu trebuie generate de programator. În schimb, într-un program trebuie specificat codul care se execută la apariția 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**. În 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 conține două butoane.

Listing 9.1: O fereastră cu două butoane

```
import java.awt.*;
public class ExempluAWT1 {
    public static void main(String args[]) {

        // Crearea ferestrei - un obiect de tip Frame
        Frame f = new Frame("O fereastră");

        // 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 ferestrei
f.show();
}
}
```

După cum veți observa la execuția acestui program, atât butoanele adăugate de noi cât și butonul de închidere a ferestrei sunt funcționale, 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 și nici pozițiile în acestea să fie plasate. Cu toate acestea ele sunt plasate unul lângă celalalt, fără să se suprapună iar suprafața ferestrei 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 și care se ocupă cu gestionarea ferestrei și cu plasarea componentelor într-o anumită ordine pe suprafața ei. Așadar, modul de aranjare nu este o caracteristică a suprafeței de afișare ci, fiecare container are asociat un obiect care se ocupă cu dimensionarea și dispunerea componentelor pe suprafața de afișare și care se numește *gestionar de poziționare* (*layout manager*) (vezi "Gestionarea poziționării").

9.2.1 Componentele AWT

După cum am spus deja, toate componentele AWT sunt definte de clase proprii ce se gasesc în pachetul `java.awt`, clasa **Component** fiind superclasa abstracta a tuturor acestor clase.

- **Button** - butoane cu eticheta formată dintr-un text pe o singură linie;
- **Canvas** - suprafață 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 și care se deschid la apăsarea lor;

- **Container** - superclasa tuturor suprafețelor de afișare (vezi "Suprafețe de afișare");
- **Label** - etichete simple ce pot conține o singură linie de text needitabil;
- **List** - liste cu selecție simplă sau multiplă;
- **Scrollbar** - bare de defilare orizontale sau verticale;
- **TextComponent** - superclasa componentelor pentru editarea textului: **TextField** (pe o singură linie) și **TextArea** (pe mai multe linii).

Mai multe informații legate de aceste clase vor fi prezentate în secțiunea "Folosirea componentelor AWT".

Din cauza unor diferențe esențiale 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, moștenite din clasa **Component**. Acestea servesc uzual pentru aflarea sau setarea atributelor obiectelor, cum ar fi: dimensiune, poziție, culoare, font, etc. și au formatul general *getProprietate*, respectiv *setProprietate*. Cele mai folosite, grupate pe tipul proprietății gestionate sunt:

- **Poziție**
getLocation, getX, getY, getLocationOnScreen
setLocation, setX, setY
- **Dimensiuni**
getSize, getHeight, getWidth
setSize, setHeight, setWidth
- **Dimensiuni și poziție**
getBounds
setBounds
- **Culoare (text și fundal)**
getForeground, getBackground
setForeground, setBackground

- **Font**
 - getFont
 - setFont
- **Vizibilitate**
 - setVisible
 - isVisible
- **Interactivitate**
 - setEnabled
 - isEnabled

9.2.2 Suprafețe de afișare (Clasa Container)

Crearea obiectelor grafice nu realizează automat și afișarea lor pe ecran. Mai întâi ele trebuie așezate pe o suprafață, care poate fi o fereastră sau suprafața unui applet, și vor deveni vizibile în momentul în care suprafața respectivă va fi vizibilă. O astfel de suprafață pe care sunt plasate componentele se numește *suprafață de afișare* sau *container* și reprezintă o instanța a unei clase derivată din **Container**. O parte din clasele a căror părinte este **Container** este prezentată mai jos:

- **Window** - este superclasa tuturor ferestrelor. Din această clasă sunt derivate:
 - **Frame** - ferestre standard;
 - **Dialog** - ferestre de dialog modale sau nemodale;
- **Panel** - o suprafață 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.

Așadar, un container este folosit pentru a adăuga componente pe suprafața lui. Componentele adăugate sunt memorate într-o listă iar pozițiile 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 adăugată pe ultima poziție a listei.

Clasa `Container` conține metodele comune tuturor suprafețelor de afișare. Dintre cele mai folosite, amintim:

- **add** - permite adăugarea unei componente pe suprafața de afișare. O componentă nu poate aparține decât unui singur container, ceea ce înseamnă că pentru a muta un obiect dintr-un container în altul trebuie să-l eliminăm mai întâi de pe containerul initial.
- **remove** - elimină o componentă de pe container;
- **setLayout** - stabilește gestionarul de poziționare al containerului (vezi "Gestionarea poziționării");
- **getInsets** - determină distanța rezervată pentru marginile suprafeței de afișare;
- **validate** - forțează containerul să reșeze toate componentele sale. Această metodă trebuie apelată explicit atunci când adăugăm sau eliminăm componente pe suprafața de afișare după ce aceasta a devenit vizibilă.

Exemplu:

```
Frame f = new Frame("O fereastră");

// Adaugam un buton direct pe fereastră
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 fereastră
// si, indirect, cele doua componente
f.add(panel);
```

9.3 Gestionarea poziționării

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

Listing 9.2: Poziționarea 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 afișata 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 afișată 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 poziționare diferiți: `GridLayout`, respectiv `FlowLayout`.

Definiție

Un *gestionar de poziționare* (*layout manager*) este un obiect care controlează dimensiunea și aranjarea (poziția) componentelor unui container.

Așadar, modul de aranjare a componentelor pe o suprafață de afișare 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 suprafața sa și anume gestionarul său de poziționare. Toate clasele care instanțiază obiecte pentru gestionarea poziționării implementează interfața `LayoutManager`.

La instanțierea unui container se creează implicit un gestionar de poziționare asociat acestuia. De exemplu, pentru o fereastră gestionarul implicit este de tip `BorderLayout`, în timp ce pentru un panel este de tip `FlowLayout`.

9.3.1 Folosirea gestionarilor de poziționare

Așa cum am văzut, orice container are un gestionar implicit de poziționare - un obiect care implementează interfața `LayoutManager`, acesta fiindu-i atașat automat la crearea sa. În cazul în care acesta nu corespunde necesităților noastre, el poate fi schimbat cu ușurință. Cei mai utilizați gestionari din pachetul `java.awt` sunt:

- `FlowLayout`
- `BorderLayout`
- `GridLayout`
- `CardLayout`
- `GridBagLayout`

Pe lângă aceștia, mai există și cei din modelul Swing care vor fi prezentați în capitolul dedicat dezvoltării de aplicații GUI folosind Swing.

Atașarea explicită a unui gestionar de poziționare la un container se face cu metoda **setLayout** a clasei **Container**. Metoda poate primi ca parametru orice instanță a unei clase care implementează interfața **LayoutManager**. Secvența de atașare 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 poziționare, dar în cazul când avem nevoie de obiectul gestionar îl putem obține cu metoda **getManager** din clasa **Container**.

Una din facilitățile cele mai utile oferite de gestionarii de poziționare este rearanjarea componentele unui container atunci când acesta este redimensionat. Pozițiile și dimensiunile componentelor nu sunt fixe, ele fiind ajustate automat de către gestionar la fiecare redimensionare astfel încât să ocupe cât mai "estetic" suprafața de afișare. Cum sunt determinate însă dimensiunile implicite ale componentelor ?

Fiecare clasă derivată din **Component** poate implementa metodele **getPreferredSize**, **getMinimumSize** și **getMaximumSize** care să returneze dimensiunea implicită a componentei respective și limitele în afara cărora componenta nu mai poate fi desenată. Gestionarii de poziționare vor apela aceste metode pentru a calcula dimensiunea la care vor afișa o componentă.

Sunt însă situații când dorim să plasăm componentele la anumite poziții fixe iar acestea să rămâna acolo chiar dacă redimensionăm containerul. Folosind un gestionar această *poziționare absolută* a componentelor nu este posibilă și deci trebuie cumva să renunțăm la gestionarea automată a containerul. Acest lucru se realizează prin trimiterea argumentului **null** metodei **setLayout**:

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

Folosind poziționarea absolută, nu va mai fi însă suficient să adăugăm cu metoda **add** componentele în container, ci va trebui să specificăm poziția și

dimensiunea lor - acest lucru era făcut automat de gestionarul de poziționare.

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

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

În general, se recomandă folosirea gestionarilor de poziționare în toate situațiile când acest lucru este posibil, deoarece permit programului să aibă aceeași "înfățișare" indiferent de platforma și rezoluția pe care este rulat. Poziționarea absolută poate ridica diverse probleme în acest sens.

Să analizăm în continuare pe fiecare din gestionarii amintiți anterior.

9.3.2 Gestionarul `FlowLayout`

Acest gestionar așează componentele pe suprafața de afișare în flux liniar, mai precis, componentele sunt adăugate una după alta pe linii, în limita spațiului disponibil. În momentul când o componentă nu mai încapă 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 și pe centru. Implicit, componentele sunt centrate pe fiecare linie iar distanța implicită între componente este de 5 pixeli pe verticală și 5 pe orizontală.

Este gestionarul implicit al containerelor derivate din clasa `Panel` deci și al applet-urilor.

Listing 9.3: Gestionarul `FlowLayout`

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

        Button b1 = new Button("Button 1");
        Button b2 = new Button("2");
        Button b3 = new Button("Button 3");
```

```

    Button b4 = new Button("Long-Named Button 4");
    Button b5 = new Button("Button 5");

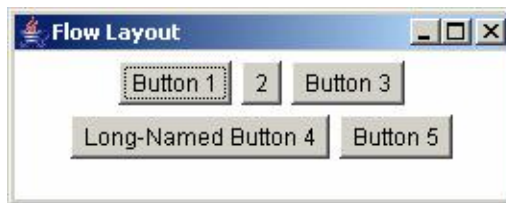
    f.add(b1); f.add(b2); f.add(b3); f.add(b4); f.add(b5);
    f.pack();
    f.show();
}
}

```

Componentele ferestrei vor fi afișate astfel:



Redimensionând fereastra astfel încât cele cinci butoane să nu mai încapă pe o linie, ultimele dintre ele vor fi trecute pe linia următoare:



9.3.3 Gestionarul BorderLayout

Gestionarul `BorderLayout` împarte suprafața de afișare în cinci regiuni, corespunzătoare celor patru puncte cardinale și centrului. O componentă poate fi plasată în oricare din aceste regiuni, dimensiunea componentei fiind calculată astfel încât să ocupe întreg spațiul de afișare 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`").

Așadar, la adăugarea unei componente pe o suprafață gestionată de `BorderLayout`, metoda `add` va mai primi pe lângă referința componentei și zona în care aceasta va fi amplasată, care va fi specificată prin una din constantele clasei: `NORTH`, `SOUTH`, `EAST`, `WEST`, `CENTER`.

`BorderLayout` este gestionarul implicit pentru toate containerele care descind din clasa `Window`, deci al tuturor tipurilor de ferestre.

Listing 9.4: Gestionarul BorderLayout

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

        f.add(new Button("Nord"), BorderLayout.NORTH);
        f.add(new Button("Sud"), BorderLayout.SOUTH);
        f.add(new Button("Est"), BorderLayout.EAST);
        f.add(new Button("Vest"), BorderLayout.WEST);
        f.add(new Button("Centru"), BorderLayout.CENTER);
        f.pack();

        f.show();
    }
}
```

Cele cinci butoane ale ferestrei vor fi afișate astfel:



La redimensionarea ferestrei se pot observa următoarele lucruri: nordul și sudul se redimensionează doar pe orizontală, estul și vestul doar pe verticală, în timp ce centrul se redimensionează atât pe orizontală cât și pe verticală. Redimensionarea componentelor din fiecare zonă se face astfel încât ele ocupă toată zona containerului din care fac parte.

9.3.4 Gestionarul GridLayout

Gestionarul `GridLayout` organizează containerul ca un tabel cu rânduri și 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 și coloane vor fi specificate în constructorul gestionarului, dar pot fi modificate

și ulterior prin metodele `setRows`, respectiv `setCols`. Dacă numărul de linii sau coloane este 0 (dar nu ambele în același timp), atunci componentele vor fi plasate într-o singură coloană sau linie. De asemenea, distanța între componente pe orizontală și distanța între rândurile tabelului pot fi specificate în constructor sau stabilite ulterior.

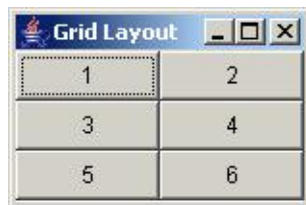
Listing 9.5: Gestionarul `GridLayout`

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

        f.add(new Button("1"));
        f.add(new Button("2"));
        f.add(new Button("3"));
        f.add(new Button("4"));
        f.add(new Button("5"));
        f.add(new Button("6"));

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

Cele șase butoane ale ferestrei vor fi plasate pe trei rânduri și două coloane, astfel:



Redimensionarea ferestrei va determina redimensionarea tuturor celulelor și deci a tuturor componentelor, atât pe orizontală cât și pe verticală.

9.3.5 Gestionarul `CardLayout`

Gestionarul `CardLayout` tratează componentele adăugate pe suprafața sa într-o manieră similară cu cea a dispunerii cărților de joc într-un pachet.

Suprafața de afișare poate fi asemănată cu pachetul de cărți 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 afișată o anumită componentă din pachet, sau să se poată parcurge secvențial pachetul, ordinea componentelor fiind internă gestionarului.

Principala utilitate a acestui gestionar este utilizarea mai eficientă a spațiului disponibil în situații în care componentele pot fi grupate în așa fel încât utilizatorul să interacționeze la un moment dat doar cu un anumit grup (o carte din pachet), celelalte fiind ascunse.

O clasă Swing care implementează un mecanism similar este `JTabbedPane`.

Listing 9.6: Gestionarul `CardLayout`

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

public class TestCardLayout extends Frame implements
    ActionListener {
    Panel tab;
    public TestCardLayout() {
        super("Test CardLayout");
        Button card1 = new Button("Card 1");
        Button card2 = new Button("Card 2");

        Panel butoane = new Panel();
        butoane.add(card1);
        butoane.add(card2);

        tab = new Panel();
        tab.setLayout(new CardLayout());

        TextField tf = new TextField("Text Field");
        Button btn = new Button("Button");
        tab.add("Card 1", tf);
        tab.add("Card 2", btn);

        add(butoane, BorderLayout.NORTH);
        add(tab, BorderLayout.CENTER);

        pack();
        show();
    }
}
```

```

        card1.addActionListener(this);
        card2.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e) {
        CardLayout gestionar = (CardLayout) tab.getLayout();
        gestionar.show(tab, e.getActionCommand());
    }

    public static void main(String args[]) {
        TestCardLayout f = new TestCardLayout();
        f.show();
    }
}

```

Prima "carte" este vizibilă



A doua "carte" este vizibilă



9.3.6 Gestionarul GridBagLayout

Este cel mai complex și flexibil gestionar de poziționare din Java. La fel ca în cazul gestionarului `GridLayout`, suprafața de afișare este considerată ca fiind un tabel însă, spre deosebire de acesta, numărul de linii și de coloane sunt determinate automat, în funcție de componentele amplasate pe suprafața de afișare. De asemenea, în funcție de componentele gestionate, dimensiunile celulelor pot fi diferite cu singurele restricții ca pe aceeași linie să aibă aceeași înălțime, iar pe coloană aibă aceeași lățime. Spre deosebire de `GridLayout`, o componentă poate ocupa mai multe celule adiacente, chiar de dimensiuni diferite, zona ocupată fiind referită prin "regiunea de afișare" a componentei respective.

Pentru a specifica modul de afișare a unei componente, acesteia îi este asociat un obiect de tip `GridBagConstraints`, în care se specifică diferite proprietăți ale componentei referitoare la regiunea să de afișare și la modul în care va fi plasată în această regiune. Legătura dintre o componentă și un obiect `GridBagConstraints` se realizează prin metoda `setConstraints`:

```
GridBagLayout gridBag = new GridBagLayout();
```

```

container.setLayout(gridBag);
GridBagConstraints c = new GridBagConstraints();
//Specificam restricțiile referitoare la afisarea componentei
. . .
gridBag.setConstraints(componenta, c);
container.add(componenta);

```

Așadar, înainte de a adăuga o componentă pe suprafața unui container care are un gestionar de tip `GridBagLayout`, va trebui să specificăm anumiți 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 aceleași constrângeri de afișare:

```

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 colțul stânga sus al componentei;
- `gridwidth`, `gridheight` - numărul de celule pe linie și coloană pe care va fi afișată componenta;
- `fill` - folosită pentru a specifica dacă o componentă va ocupa întreg spațiul pe care îl are destinat; valorile posibile sunt `HORIZONTAL`, `VERTICAL`, `BOTH`, `NONE`;
- `insets` - distanțele dintre componentă și marginile suprafeței sale de afișare;
- `anchor` - folosită atunci când componenta este mai mică decât suprafața sa de afișare pentru a forța o anumită dispunere a sa: nord, sud, est, vest, etc.
- `weightx`, `weighty` - folosite pentru distribuția spațiului 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` și adăugarea unei componente cu restricțiile stabilite pe fereastră.



Listing 9.7: Gestionarul GridBagLayout

```
import java.awt.*;

public class TestGridBagLayout {
    static Frame f;
    static GridBagLayout gridBag;
    static GridBagConstraints gbc;

    static void adauga(Component comp,
        int x, int y, int w, int h) {
        gbc.gridx = x;
        gbc.gridy = y;
        gbc.gridwidth = w;
        gbc.gridheight = h;

        gridBag.setConstraints(comp, gbc);
        f.add(comp);
    }

    public static void main(String args[]) {

        f = new Frame("Test GridBagLayout");
        gridBag = new GridBagLayout();

        gbc = new GridBagConstraints();
        gbc.weightx = 1.0;
        gbc.weighty = 1.0;
```

```
gbc.insets = new Insets(5, 5, 5, 5);

f.setLayout(gridBag);

Label mesaj = new Label("Evidenta persoane", Label.CENTER
    );
mesaj.setFont(new Font("Arial", Font.BOLD, 24));
mesaj.setBackground(Color.yellow);
gbc.fill = GridBagConstraints.BOTH;
adauga(mesaj, 0, 0, 4, 2);

Label etNume = new Label("Nume:");
gbc.fill = GridBagConstraints.NONE;
gbc.anchor = GridBagConstraints.EAST;
adauga(etNume, 0, 2, 1, 1);

Label etSalariu = new Label("Salariu:");
adauga(etSalariu, 0, 3, 1, 1);

TextField nume = new TextField("", 30);
gbc.fill = GridBagConstraints.HORIZONTAL;
gbc.anchor = GridBagConstraints.CENTER;
adauga(nume, 1, 2, 2, 1);

TextField salariu = new TextField("", 30);
adauga(salariu, 1, 3, 2, 1);

Button adaugare = new Button("Adaugare");
gbc.fill = GridBagConstraints.NONE;
adauga(adaugare, 3, 2, 1, 2);

Button salvare = new Button("Salvare");
gbc.fill = GridBagConstraints.HORIZONTAL;
adauga(salvare, 1, 4, 1, 1);

Button iesire = new Button("Iesire");
adauga(iesire, 2, 4, 1, 1);

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

9.3.7 Gruparea componentelor (Clasa Panel)

Plasarea componentelor direct pe suprafața de afișare poate deveni incomodă în cazul în care avem multe obiecte grafice. Din acest motiv, se recomandă gruparea componentelor înrudite ca funcții astfel încât să putem fi siguri că, indiferent de gestionarul de poziționare al suprafeței de afișare, 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 suprafață de afișare pentru componente grafice, inclusiv pentru alte panel-uri. Clasa care instanțiază 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 poziționare anume, folosind metoda `setLayout`. Gestionarul implicit pentru containerele de tip **Panel** este **FlowLayout**.

Așadar, o aranjare eficientă a componentelor unei ferestre înseamnă:

- gruparea componentelor ”înfrățite” (care nu trebuie să fie despartite de gestionarul de poziționare al ferestrei) în panel-uri;
- aranjarea componentelor unui panel, prin specificarea unui gestionar de poziționare corespunzător;
- aranjarea panel-urilor pe suprafața ferestrei, prin specificarea gestionarului de poziționare al ferestrei.

Listing 9.8: Gruparea componentelor

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

        Panel intro = new Panel();
        intro.setLayout(new GridLayout(1, 3));
        intro.add(new Label("Text:"));
        intro.add(new TextField("", 20));
        intro.add(new Button("Adaugare"));

        Panel lista = new Panel();
        lista.setLayout(new FlowLayout());
        lista.add(new List(10));
        lista.add(new Button("Stergere"));
    }
}
```

```

Panel control = new Panel();
control.add(new Button("Salvare"));
control.add(new Button("Iesire"));

f.add(intro, BorderLayout.NORTH);
f.add(lista, BorderLayout.CENTER);
f.add(control, BorderLayout.SOUTH);

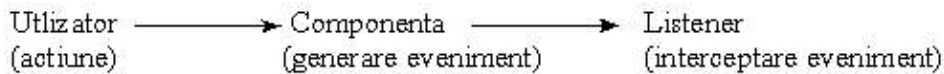
f.pack();
f.show();
}
}

```

9.4 Tratarea evenimentelor

Un *eveniment* este produs de o acțiune a utilizatorului asupra unei componente grafice și 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 și *surse de evenimente*.

Interceptarea evenimentelor generate de componentele unui program se realizează prin intermediul unor clase de tip **listener** (ascultător, consumator de evenimente). În Java, orice obiect poate "consuma" evenimentele generate de o anumită componentă grafică.



Așadar, pentru a scrie cod care să se execute în momentul în care utilizatorul interacționează cu o componentă grafică trebuie să facem următoarele lucruri:

- să scriem o clasă de tip listener care să "asculte" evenimentele produse de acea componentă și în cadrul acestei clase să implementăm metode specifice pentru tratarea lor;

- să comunicăm componentei sursă că respectiva clasă 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 funcție de componenta care le generează, mai precis în funcție de acțiunea utilizatorului asupra acesteia. Pentru fiecare tip de eveniment există o clasă care instanțiază obiecte de acel tip. De exemplu, evenimentul generat de acționarea 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 declarația sa că dorește să asculte evenimente de un anumit tip. Acest lucru se realizează prin implementarea unei interfețe specifice fiecărui tip de eveniment. Astfel, pentru ascultarea evenimentelor de tip `ActionEvent` clasa respectivă trebuie să implementeze interfața `ActionListener`, pentru `TextEvent` interfață care trebuie implementată este `TextListener`, etc. Toate aceste interfețe sunt derivate din **EventListener**.

Fiecare interfață definește una sau mai multe metode care vor fi apelate automat la apariția unui eveniment:

```
class AscultaButoane implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        // Metoda interfeței ActionListener
        ...
    }
}
class AscultaTexte implements TextListener {
    public void textValueChanged(TextEvent e) {
        // Metoda interfeței TextListener
        ...
    }
}
```

Intrucât o clasă poate implementa oricâte interfețe, 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 interfețe pentru a ști ce trebuie să implementeze o clasă consumatoare de evenimente.

Așa cum am spus mai devreme, pentru ca evenimentele unei componente să fie interceptate de către o instanță a unei clase ascultător, această clasă trebuie înregistrată în lista ascultătorilor componentei respective. Am spus lista, deoarece evenimentele unei componente pot fi ascultate de oricâte clase, cu condiția ca acestea să fie înregistrate la componenta respectivă. Înregistrarea unei clase în lista ascultătorilor unei componente se face cu metode din clasa `Component` de tipul **`addTipEvenimentListener`**, iar eliminarea ei din această listă cu **`removeTipEvenimentListener`**.

Sumarizând, tratarea evenimentelor în Java se desfășoară 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 interfețe specifice fiecărui tip de eveniment - acestea descriu metode ce vor fi apelate automat la apariția evenimentelor.

9.4.1 Exemplu de tratare a evenimentelor

Înainte de a detalia aspectele prezentate mai sus, să considerăm un exemplu de tratare a evenimentelor. Vom crea o fereastră care să conțină două butoane cu numele "OK", respectiv "Cancel". La apăsarea fiecărui buton vom scrie pe bara de titlu a ferestrei mesajul "Ati apasat butonul ...".

Listing 9.9: Ascultarea evenimentelor a două butoane

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

```

class Fereastra extends Frame {
    public Fereastra(String titlu) {
        super(titlu);
        setLayout(new FlowLayout());
        setSize(200, 100);
        Button b1 = new Button("OK");
        Button b2 = new Button("Cancel");
        add(b1);
        add(b2);

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

class Ascultator implements ActionListener {
    private Fereastra f;
    public Ascultator(Fereastra f) {
        this.f = f;
    }

    // Metoda interfetei ActionListener
    public void actionPerformed(ActionEvent e) {
        f.setTitle("Ati apasat " + e.getActionCommand());
    }
}

public class TestEvent1 {
    public static void main(String args[]) {
        Fereastra f = new Fereastra("Test Event");
        f.show();
    }
}

```

Nu este obligatoriu să definim clase speciale pentru ascultarea evenimentelor. In exemplul de mai sus am definit clasa `Ascultator` pentru a intercepta evenimentele produse de cele două butoane și din acest motiv a trebuit să trimitem ca parametru constructorului clasei o referință la fereastra noastră. Mai simplu ar fi fost să folosim chiar clasa `Fereastra` pentru a trata evenimentele produse de componentele sale. Vom modifica puțin și

aplicația pentru a pune în evidența o altă modalitate de a determina componenta generatoare a unui eveniment - metoda **getSource**.

Listing 9.10: Tratarea evenimentelor în fereastră

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

class Fereastră extends Frame implements ActionListener {
    Button ok = new Button("OK");
    Button exit = new Button("Exit");
    int n=0;

    public Fereastră(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 Fereastră
        // 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[]) {
        Fereastră f = new Fereastră("Test Event");
        f.show();
    }
}
```

Așadar, orice clasă poate asculta evenimente de orice tip cu condiția să implementeze interfețele specifice acelor tipuri de evenimente.

9.4.2 Tipuri de evenimente

Evenimentele se împart în două categorii: *de nivel jos* și *semantice*.

Evenimentele de nivel jos reprezintă o interacțiune de nivel jos cum ar fi o apăsare de tastă, mișcarea mouse-ului, sau o operație asupra unei ferestre. În tabelul de mai jos sunt enumerate clasele ce descriu aceste evenimente și operațiunile efectuate (asupra unei componente) care le generează:

<code>ComponentEvent</code>	Ascundere, deplasare, redimensionare, afișare
<code>ContainerEvent</code>	Adăugare pe container, eliminare
<code>FocusEvent</code>	Obținere, pierdere focus
<code>KeyEvent</code>	Apăsare, eliberare taste, tastare
<code>MouseEvent</code>	Operațiuni cu mouse-ul: click, drag, etc.
<code>WindowEvent</code>	Operațiuni asupra ferestrelor: minimizare, maximizare, etc.

O anumită acțiune a utilizatorului poate genera mai multe evenimente. De exemplu, tastarea literei 'A' va genera trei evenimente: unul pentru apăsare, unul pentru eliberare și unul pentru tastare. În funcție de necesitățile aplicației putem scrie cod pentru tratarea fiecărui eveniment în parte.

Evenimentele semantice reprezintă interacțiunea cu o componentă GUI: apăsarea unui buton, selectarea unui articol dintr-o listă, etc. Clasele care descriu aceste tipuri de evenimente sunt:

<code>ActionEvent</code>	Acționare
<code>AdjustmentEvent</code>	Ajustarea unei valori
<code>ItemEvent</code>	Schimbarea stării
<code>TextEvent</code>	Schimbarea textului

Următorul tabel prezintă componentele AWT și tipurile de evenimente generate, prezentate sub forma interfețelor corespunzătoare. Evident, evenimentele generate de o superclasă, cum ar fi `Component`, se vor regăsi și pentru toate subclasele sale.

<code>Component</code>	<code>ComponentListener</code> <code>FocusListener</code> <code>KeyListener</code> <code>MouseListener</code> <code>MouseMotionListener</code>
<code>Container</code>	<code>ContainerListener</code>
<code>Window</code>	<code>WindowListener</code>
<code>Button</code> <code>List</code> <code>MenuItem</code> <code>TextField</code>	<code>ActionListener</code>
<code>Choice</code> <code>Checkbox</code> <code>List</code> <code>CheckboxMenuItem</code>	<code>ItemListener</code>
<code>Scrollbar</code>	<code>AdjustmentListener</code>
<code>TextField</code> <code>TextArea</code>	<code>TextListener</code>

Observați că deși există o singură clasă `MouseEvent`, există două interfețe asociate `MouseListener` și `MouseMotionListener`. Acest lucru a fost făcut deoarece evenimentele legate de deplasarea mouse-ului sunt generate foarte frecvent și recepționarea lor poate avea un impact negativ asupra vitezei de execuție, în situația când tratarea acestora nu ne interesează și dorim să tratăm doar evenimente de tip click, de exemplu.

Orice clasă care tratează evenimente trebuie să implementeze obligatoriu metodele interfețelor corespunzătoare. Tabelul de mai jos prezintă, pentru fiecare interfață, metodele puse la dispoziție și care trebuie implementate de către clasa ascultător.

Interfață	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)

În cazul în care un obiect listener tratează evenimente de același 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 reacționa în consecință. Toate tipurile de evenimente moștenesc metoda **getSource** care returnează obiectul responsabil cu generarea evenimentului. În cazul în care dorim să diferențiem 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 dispoziție și alte metode specifice care permit aflarea de informații legate de evenimentul generat. De exemplu, `ActionEvent` conține metoda `getActionCommand` care, implicit, returnează eticheta butonului care a fost apăsat. Astfel de particularități vor fi prezentate mai detaliat în secțiunile dedicate fiecărei componente în parte.

9.4.3 Folosirea adaptorilor și a claselor anonime

Am vazut că o clasă care tratează evenimente de un anumit tip trebuie să implementeze interfața corespunzătoare aceluși tip. Aceasta înseamnă că trebuie să implementeze obligatoriu toate metodele definite de acea interfață, chiar dacă nu specifică nici un cod pentru unele dintre ele. Sunt însă situații când acest lucru poate deveni supărător, mai ales atunci când nu ne interesează decât o singură metodă a interfeței.

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 colțul dreapta sus și nici cu combinația de taste Alt+F4. Pentru a realiza acest lucru trebuie interceptat evenimentul de închidere a ferestrei în metoda `windowClosing` și apelată metoda `dispose` de închidere a ferestrei, sau `System.exit` pentru terminarea programului, în cazul când este vorba de fereastra principală a aplicației. Aceasta înseamnă că trebuie să implementăm interfața `WindowListener` care are nu mai puțin de **sapte** metode.

Listing 9.11: Implementarea interfeței `WindowListener`

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

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

    // Metodele interfeței 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[]) {
        Fereastră f = new Fereastră("Test WindowListener");
        f.show();
    }
}
```

Observați că trebuie să implementăm toate metodele interfeței, 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 dorește să închidă fereastra. Pentru a evita scrierea inutilă a

acestor metode, există o serie de clase care implementează interfețele de tip "listener" fără a specifica nici un cod pentru metodele lor. Aceste clase se numesc **adaptori**.

Un *adapter* este o clasă abstractă care implementează o anumită interfață fără a specifica cod nici unei metode a interfeței.

Scopul unei astfel de clase este ca la crearea unui "ascultător" de evenimente, în loc să implementă o anumită interfață și implicit toate metodele sale, să extindem adaptorul corespunzător interfeței respective (dacă are!) și să supradefinim doar metodele care ne interesează (cele în care vrem să scriem o anumită secvență de cod).

De exemplu, adaptorul interfeței `WindowListener` este `WindowAdapter` iar folosirea acestuia este dată în exemplul de mai jos:

Listing 9.12: Extinderea clasei `WindowAdapter`

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

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

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

public class TestWindowAdapter {
    public static void main(String args[]) {
        Fereastră f = new Fereastră("Test WindowAdapter");
        f.show();
    }
}
```

Avantajul clar al acestei modalități de tratare a evenimentelor este reducerea codului programului, acesta devenind mult mai lizibil. Însă există și două dezavantaje majore. După cum ați observat față de exemplul anterior,

clasa `Fereastra` nu poate extinde `WindowAdapter` deoarece ea extinde deja clasa `Frame` și 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 greșeală de sintaxă în declararea unei metode a interfeței nu va produce o eroare de compilare dar nici nu va supradefini metoda interfeței ci, pur și 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);
    }
}
```

În tabelul de mai jos sunt dați toți adaptorii interfețelor de tip "listener" - se observă că o interfață `XXXListener` are un adaptor de tipul `XXXAdapter`. Interfețele care nu au un adaptor sunt cele care definesc o singură metodă și prin urmare crearea unei clase adaptor nu își are rostul.

Interfața	Adaptor
<code>ActionListener</code>	nu are
<code>AdjustmentListener</code>	nu are
<code>ComponentListener</code>	<code>ComponentAdapter</code>
<code>ContainerListener</code>	<code>ContainerAdapter</code>
<code>FocusListener</code>	<code>FocusAdapter</code>
<code>ItemListener</code>	nu are
<code>KeyListener</code>	<code>KeyAdapter</code>
<code>MouseListener</code>	<code>MouseAdapter</code>
<code>MouseMotionListener</code>	<code>MouseMotionAdapter</code>
<code>TextListener</code>	nu are
<code>WindowListener</code>	<code>WindowAdapter</code>

Știm că o clasă internă este o clasă declarată în cadrul altei clase, iar clasele anonime sunt clase interne folosite pentru instanțierea unui singur obiect de un anumit tip. Un exemplu tipic de folosire a lor este instanțierea

adaptorilor direct în corpul unei clase care conține componente ale căror evenimente trebuie tratate.

Listing 9.13: Folosirea adaptorilor și a claselor anonime

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

class Fereastră extends Frame {
    public Fereastră(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 = Fereastră.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 = Fereastră.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 adaptorii");
        f.show();
    }
}
```

9.5 Folosirea ferestrelor

După cum am văzut suprafețele de afișare 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** și **Dialog**.

O aplicație Java cu interfață grafică va fi formată din una sau mai multe ferestre, una dintre ele fiind numită *fereastra principală*.

9.5.1 Clasa Window

Clasa **Window** este rar utilizată în mod direct deoarece permite doar crearea unor ferestre care nu au chenar și nici bară de meniuri. Este utilă atunci când dorim să afișăm ferestre care nu interacționează cu utilizatorul ci doar oferă anumite informații.

Metodele mai importante ale clasei **Window**, care sunt de altfel moștenite 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 vizibilă se poate apela metoda **show**;

- `isShowing` - testează dacă fereastra este vizibilă sau nu;
- `dispose` - închide) fereastra și și eliberează toate resursele acesteia;
- `pack` - redimensionează automat fereastra la o suprafața optimă care să cuprindă toate componentele sale; trebuie apelată în general după adăugarea tuturor componentelor pe suprafața ferestrei.
- `getFocusOwner` - returnează componenta ferestrei care are focus-ul (dacă fereastra este activă).

9.5.2 Clasa Frame

Este derivată a clasei `Window` și este folosită pentru crearea de ferestre independente și funcționale, eventual conținând o bară de meniuri. Orice aplicație cu interfață grafică conține cel puțin o fereastră, cea mai importantă fiind numită și *fereastră principală*.

Constructorii uzuali ai clasei `Frame` permit crearea unei ferestre cu sau fără titlu, inițial 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 instanțierea directă a obiectelor de tip `Frame` este mai puțin 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 Fereastră extends Frame{
    // Constructorul
    public Fereastră(String titlu) {
        super(titlu);
        ...
    }
}
```

```

}
public class TestFrame {
    public static void main(String args[]) {
        Fereastra f = new Fereastra("Titlul ferestrei");
        f.show();
    }
}

```

Gestionarul de poziționare implicit al clasei `Frame` este `BorderLayout`. Din acest motiv, în momentul în care fereastra este creată dar nici o componentă grafică nu este adăugată, suprafața de afișare a ferestrei va fi determinată automota de gestionarul de poziționare și va oferi doar spațiul necesar afișării barei ferestrei și grupului de butoane pentru minimizare, maximizare și închidere. Același efect îl vom obține dacă o redimensionăm și apelăm apoi metoda `pack` care determină dimensiunea suprafeței de afișare în funcție de componentele adăugate.

Se observă de asemenea că butonul de închidere a ferestrei nu este funcțional. Tratarea evenimentelor ferestrei se face prin implementarea interfeței `WindowListener` sau, mai uzual, prin folosirea unui adaptor de tip `WindowAdapter`.

Structura generală a unei ferestre este descrisă de clasa `Fereastra` din exemplul de mai jos:

Listing 9.14: Structura generală a unei ferestre

```

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

class Fereastra extends Frame implements ActionListener {

    // Constructorul
    public Fereastra(String titlu) {
        super(titlu);

        // Tratăm evenimentul de închidere a ferestrei
        this.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                dispose(); // închidem fereastra
                // sau terminăm aplicația
                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 moștenite 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 aplicații;
- `setIconImage` - setează iconița ferestrei;
- `setMenuBar` - setează bara de meniuri a ferestrei (vezi "Folosirea meniurilor");
- `setTitle` - setează titlul ferestrei;

- `setResizable` - stabilește dacă fereastra poate fi redimensionată de utilizator;

9.5.3 Clasa Dialog

Toate interfețele grafice oferă un tip special de ferestre destinate preluării unor informații sau a unor date de la utilizator. Acestea se numesc *ferestre de dialog* sau *casete de dialog* și sunt implementate prin intermediul clasei **Dialog**, subclasă directă a clasei **Window**.

Diferența majoră dintre ferestrele de dialog și ferestrele de tip **Frame** constă în faptul că o fereastră de dialog este dependentă de o altă fereastră (normală sau tot fereastră de dialog), numită și *fereastră părinte*. Cu alte cuvinte, ferestrele de dialog nu au o existență de sine stătătoare. Când fereastra părinte este distrusă sunt distruse și 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 părinte în momentul deschiderii lor, cum ar fi ferestrele de introducere a unor date, de alegere a unui fișier, de selectare a unei opțiuni, mesaje de avertizare, etc;
- **nemodale**: care nu blochează fluxul de intrare către fereastra părinte - de exemplu, dialogul de căutare a unui cuvânt într-un fișier, etc.

Implicit, o fereastră de dialog este nemodală și invizibilă, însă există constructori care să specifice și acești 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 referința 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 simplă și se realizează prin derivarea clasei `Dialog`. Comunicarea dintre fereastra de dialog și fereastra sa părinte, pentru ca aceasta din urmă să poată folosi datele introduse (sau opțiunea specificată) 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 suprafața sa și 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 șir de caractere. Fereastra principală a aplicației va fi părintele casetei de dialog, va primi șirul de caractere introdus și își 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:



Listing 9.15: Folosirea unei ferestre de dialog

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

// Fereastra principala
class FerPrinc extends Frame implements ActionListener{
```

```

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

    setLayout(new FlowLayout());
    setSize(300, 80);
    Button b = new Button("Schimba titlul");
    add(b);
    b.addActionListener(this);
}

public void actionPerformed(ActionEvent e) {
    FerDialog d = new FerDialog(this, "Dati titlul", true);
    String titlu = d.raspuns;
    if (titlu == null)
        return;
    setTitle(titlu);
}
}

// Fereastra de dialog
class FerDialog extends Dialog implements ActionListener {
    public String raspuns = null;
    private TextField text;
    private Button ok, cancel;

    public FerDialog(Frame parinte, String titlu, boolean
        modala) {
        super(parinte, titlu, modala);
        this.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                raspuns = null;
                dispose();
            }
        });

        text = new TextField("", 30);
        add(text, BorderLayout.CENTER);

        Panel panel = new Panel();
        ok = new Button("OK");

```

```
cancel = new Button("Cancel");
panel.add(ok);
panel.add(cancel);

add(panel, BorderLayout.SOUTH);
pack();

text.addActionListener(this);
ok.addActionListener(this);
cancel.addActionListener(this);

show();
}

public void actionPerformed(ActionEvent e) {
    Object sursa = e.getSource();
    if (sursa == ok || sursa == text)
        raspuns = text.getText();
    else
        raspuns = null;
    dispose();
}
}

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

9.5.4 Clasa FileDialog

Pachetul `java.awt` pune la dispozitie și un tip de fereastră de dialog folosită pentru selectarea unui nume de fișier în vederea încărcării sau salvării unui fișier: clasa `FileDialog`, derivată din `Dialog`. Instanțele acestei clase au un comportament comun dialogurilor de acest tip de pe majoritatea platformelor de lucru, dar forma în care vor fi afișate este specifică platformei pe care rulează aplicația.

Constructorii clasei sunt:

```
FileDialog(Frame parinte)
```



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

Parametrul "părinte" reprezintă referința ferestrei părinte, "titlu" reprezintă titlul ferestrei iar prin argumentul "mod" specificăm dacă încercăm sau salvăm un fișier; 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 afișarea sa se face cu `show`, caseta de dialog va fi modală. Dacă afișarea se face cu `setVisible(true)`, atunci va fi nemodală. După selectarea unui fișier ea va fi făcută automat invizibilă.

Pe lângă metodele moștenite de la superclasa `Dialog` clasa `FileDialog` mai conține metode pentru obținerea numelui fișierului 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 fișier cu extensia "java". Directorul inițial este directorul curent, iar numele implicit este `TestFileDialog.java`. Numele fișierului ales va fi afișat la consolă.

Listing 9.16: Folosirea unei ferestre de dialog

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

class FerPrinc extends Frame implements ActionListener{

    public FerPrinc(String titlu) {
        super(titlu);
        this.addWindowListener(new WindowAdapter() {
```

```

        public void windowClosing(WindowEvent e) {
            System.exit(0);
        }
    });

    Button b = new Button("Alege fisier");
    add(b, BorderLayout.CENTER);
    pack();

    b.addActionListener(this);
}

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

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

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

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

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

```

Clasa `FileDialog` este folosită mai rar deoarece în Swing există clasa `JFileChooser` care oferă mai multe facilități și prin urmare va constitui prima opțiune într-o aplicație cu interfață grafică.

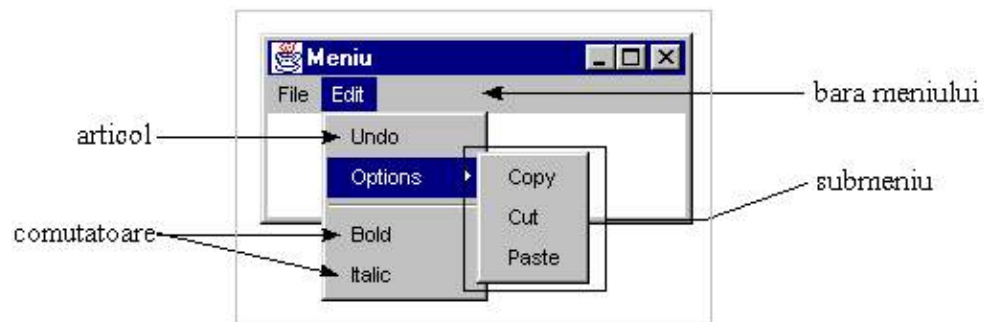
9.6 Folosirea meniurilor

Spre deosebire de celelalte obiecte grafice care derivă din clasa `Component`, componentele unui meniu reprezintă instanțe ale unor clase derivate din superclasa abstractă `MenuComponent`. Această excepție este făcută deoarece unele platforme grafice limitează capacitățile unui meniu.

Meniurile pot fi grupate în două categorii:

- **Meniuri fixe** (vizibile permanent): sunt grupate într-o bară de meniuri ce conține câte un meniu pentru fiecare intrare a sa. La rândul lor, aceste meniuri conțin articole ce pot fi selectate, comutatoare sau alte meniuri (submeniuri). O fereastră poate avea un singur meniu fix.
- **Meniuri de context** (popup): sunt meniuri invizibile asociate unei ferestre și care se activează uzual prin apăsarea butonului drept al mouse-ului. O altă diferență față de meniurile fixe constă în faptul că meniurile de context nu sunt grupate într-o bară de meniuri.

În figura de mai jos este pusă în evidență alcătuirea unui meniu fix:



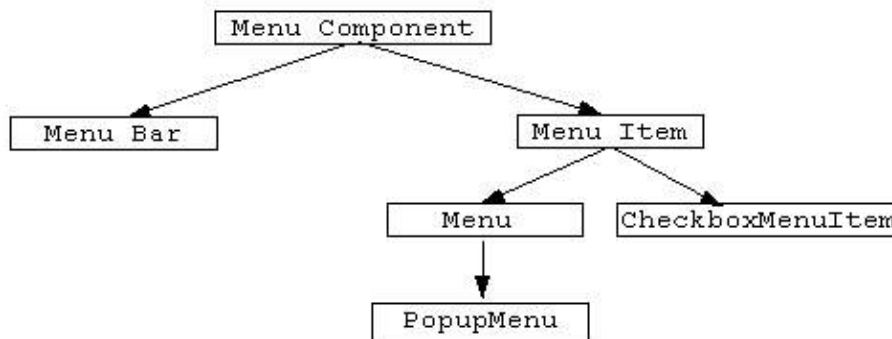
Exemplul de mai sus conține o bară de meniuri formată din două meniuri principale *File* și *Edit*. Meniul *Edit* conține la rândul lui alt meniu (submeniu) *Options*, articolul *Undo* și două comutatoare *Bold* și *Italic*. Prin abuz de limbaj, vom referi uneori bara de meniuri a unei ferestre ca fiind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

În modelul AWT obiectele care reprezintă bare de meniuri sunt reprezentate ca instanțe ale clasei `MenuBar`. Un obiect de tip `MenuBar` conține obiecte de tip `Menu`, care sunt de fapt meniurile derulante propriu-zise. La rândul lor, acestea pot conține obiecte de tip `MenuItem`, `CheckboxMenuItem`, dar și alte obiecte de tip `Menu` (submeniuri).

Pentru a putea conține un meniu, o componentă trebuie să implementeze interfața **MenuContainer**. Cel mai adesea, meniurile sunt atașate ferestrelor, mai precis obiectelor de tip **Frame**, acestea implementând interfața **MenuContainer**. Atașarea unei bare de meniuri la o fereastră se face prin metoda **addMenuBar** a clasei **Frame**.

9.6.1 Ierarhia claselor ce descriu meniuri

Să vedem în continuare care este ierarhia claselor folosite în lucrul cu meniuri și să analizăm pe rând aceste clase:



Clasa **MenuComponent** este o clasă abstractă din care sunt extinse toate celelalte clase folosite la crearea de meniuri, fiind similară celeilalte superclase abstracte **Component**. **MenuComponent** conține metode cu caracter general, dintre care amintim **getName**, **setName**, **getFont**, **setFont**, cu sintaxa și semnificațiile 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 fereastră 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 fereastră
Frame f = new Frame("Fereastră cu meniu");
f.addMenuBar(mb);
```

Orice articol al unui meniu trebuie să fie o instanță a clasei **MenuItem**. Obiectele acestei clase descriu așadar opțiunile individuale ale meniurilor derulante, cum sunt "Open", "Close", "Exit", etc. O instanță a clasei **MenuItem** reprezintă de fapt un buton sau un comutator, cu o anumită etichetă care va apărea în meniu, însoțită eventual de un accelerator (obiect de tip **MenuShortcut**) ce reprezintă combinația de taste cu care articolul poate fi apelat rapid (vezi "Acceleratori").

Clasa **Menu** permite crearea unui meniu derulant într-o bară de meniuri. Opțional, un meniu poate fi declarat ca fiind *tear-off*, ceea ce înseamnă că poate fi deschis și deplasat cu mouse-ul (dragged) într-o altă poziție decât cea originală ("rupt" din poziția sa). Acest mecanism este dependent de platformă și poate fi ignorat pe unele dintre ele. Fiecare meniu are o etichetă, care este de fapt numele său ce va fi afișat pe bara de meniuri. Articolele dintr-un meniu trebuie să aparțină clasei **MenuItem**, ceea ce înseamnă că pot fi instanțe 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), acționarea articolului determinând trecerea sa dintr-o stare în alta. La validarea unui comutator în dreptul etichetei sale va fi afișat un simbol grafic care indică acest lucru; la invalidarea sa, simbolul grafic respectiv va dispărea. Clasa **CheckboxMenuItem** are aceeași funcționalitate cu cea a casetelor de validare de tip **Checkbox**, ambele implementând interfața **ItemSelectable**.

Să vedem în continuare cum ar arăta un program care construiește un meniu ca în figura prezentată anterior:

Listing 9.17: Crearea unui meniu

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

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

        MenuBar mb = new MenuBar();

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

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

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

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

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

        f.setMenuBar(mb);
        f.setSize(200, 100);
        f.show();
    }
}
```

9.6.2 Tratarea evenimentelor generate de meniuri

La alegerea unei opțiuni dintr-un meniu se generează fie un eveniment de tip **ActionEvent** dacă articolul respectiv este de tip **MenuItem**, fie **ItemEvent** pentru comutatoarele **CheckboxMenuItem**. Așadar, pentru a activa opțiunile unui meniu trebuie implementate interfațele **ActionListener** sau/și **ItemListener** în cadrul obiectelor care trebuie să specifice codul ce va fi executat la alegerea unei opțiuni și implementate metodele **actionPerformed**, respectiv **itemStateChanged**. Fiecărui meniu îi putem asocia un obiect receptor diferit, ceea ce ușurează munca în cazul în care ierarhia de meniuri este complexă. Pentru a realiza legătura între obiectul meniu și obiectul de tip listener trebuie să adăugăm receptorul în lista de ascultători a meniului respectiv, întocmai ca pe orice componentă, folosind metodele **addActionListener**, respectiv **addItemListener**.

Așadar, tratarea evenimentelor generate de obiecte de tip **MenuItem** este identică cu tratarea butoanelor, ceea ce face posibil ca unui buton de pe suprafața de afișare să îi corespundă o opțiune dintr-un meniu, ambele cu același nume, tratarea evenimentului corespunzător apăsării butonului, sau alegerii opțiunii, făcându-se o singură dată într-o clasă care este înregistrată ca receptor atât la buton cât și la meniu.

Obiectele de tip **CheckboxMenuItem** tip se găsesc într-o categorie comună cu **List**, **Choice**, **CheckBox**, toate implementând interfața **ItemSelectable** și deci tratarea lor va fi făcută la fel. Tipul de operație selectare / deselectare este codificat în evenimentul generat de câmpurile statice **ItemEvent.SELECTED** și **ItemEvent.DESELECTED**.

Listing 9.18: Tratarea evenimentelor unui meniu

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

public class TestMenuEvent extends Frame
    implements ActionListener, ItemListener {

    public TestMenuEvent(String titlu) {
        super(titlu);

        MenuBar mb = new MenuBar();
        Menu test = new Menu("Test");
        CheckboxMenuItem check = new CheckboxMenuItem("Check me")
        ;
    }
}
```

```

    test.add(check);
    test.addSeparator();
    test.add(new MenuItem("Exit"));

    mb.add(test);
    setMenuBar(mb);

    Button btnExit = new Button("Exit");
    add(btnExit, BorderLayout.SOUTH);
    setSize(300, 200);
    show();

    test.addActionListener(this);
    check.addItemListener(this);
    btnExit.addActionListener(this);
}

public void actionPerformed(ActionEvent e) {
    // Valabila si pentru meniu si pentru buton
    String command = e.getActionCommand();
    if (command.equals("Exit"))
        System.exit(0);
}

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

public static void main(String args[]) {
    TestMenuEvent f = new TestMenuEvent("Tratare evenimente
        meniuri");
    f.show();
}
}

```

9.6.3 Meniuri de context (popup)

Au fost introduse începând cu AWT 1.1 și 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

afișate la poziția la care se găsea mouse-ul în momentul apăsării butonului său drept. Metodele de adăugare a articolelor unui meniu de context sunt moștenite 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"));
```

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

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

și este de obicei rezultatul apăsării unui buton al mouse-ului, pentru a avea acces rapid la meniu. Argumentul "origine" reprezintă componenta față de originile căreia se va calcula poziția de afișare a meniului popup. De obicei, reprezintă instanța ferestrei în care se va afișa meniul. Deoarece interacțiunea 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` și este definită în clasa `MouseEvent`. Poziționarea și afișarea meniului este însă responsabilitatea programatorului.

Meniurile de context nu se adaugă la un alt meniu (bară sau sub-meniu) ci se atașează la o componentă (de obicei la o fereastră) prin metoda **add** a acesteia. În cazul când avem mai multe meniuri popup pe care vrem să le folosim într-o fereastră, trebuie să le definim pe toate și, la un moment dat, vom adăuga ferestrei meniul corespunzător după care îl vom face vizibil. După închiderea acestuia, vom "rupe" legătura între fereastră și meniu prin instrucțiunea **remove**:

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

În exemplul de mai jos, vom crea un meniu de context pe care îl vom activa la apăsarea butonului drept al mouse-ului pe suprafața ferestrei principale. Tratarea evenimentelor generate de un meniu popup se realizează identic ca pentru meniurile fixe.

Listing 9.19: Folosirea unui meniu de context (popup)

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

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

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

        this.addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent e) {
                if (e.isPopupTrigger())
                    popup.show(origin, e.getX(), e.getY());
            }
            public void mouseReleased(MouseEvent e) {
                if (e.isPopupTrigger())
                    popup.show(origin, e.getX(), e.getY());
            }
        });
        setSize(300, 300);

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

    public void actionPerformed(ActionEvent e) {
        String command = e.getActionCommand();
        if (command.equals("Exit"))
            System.exit(0);
    }
}
```

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

9.6.4 Acceleratori (Clasa MenuItemShortcut)

Pentru articolele unui meniu este posibilă specificarea unor combinații de taste numite *acceleratori* (shortcuts) care să permită accesarea directă, prin intermediul tastaturii, a opțiunilor dintr-un meniu. Astfel, oricărui obiect de tip `MenuItem` îi poate fi asociat un obiect de tip accelerator, definit prin intermediul clasei `MenuItemShortcut`. Singurele combinații 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, MenuItemShortcut accelerator)`, ca în exemplele de mai jos:

```
// Ctrl+O  
new MenuItem("Open", new MenuItemShortcut(KeyEvent.VK_O));  
  
// Ctrl+P  
new MenuItem("Print", new MenuItemShortcut('p'));  
  
// Ctrl+Shift+P  
new MenuItem("Preview", new MenuItemShortcut('p'), true);
```

9.7 Folosirea componentelor AWT

În continuare vor fi date exemple de folosire ale componentelor AWT, în care să fie puse în evidență cât mai multe din particularitățile acestora, precum și modul de tratare a evenimentelor generate.

9.7.1 Clasa Label

Un obiect de tip `Label` (etichetă) reprezintă o componentă pentru plasarea unui text pe o suprafața de afișare. O etichetă este formată dintr-o singură linie de text static ce nu poate fi modificat de către utilizator, dar poate fi modificat din program.



Listing 9.20: Folosirea clasei `Label`

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

        Label nord, sud, est, vest, centru;

        nord = new Label("Nord", Label.CENTER);
        nord.setForeground(Color.blue);

        sud = new Label("Sud", Label.CENTER);
        sud.setForeground(Color.red);

        vest = new Label("Vest", Label.LEFT);
        vest.setFont(new Font("Dialog", Font.ITALIC, 14));

        est = new Label("Est", Label.RIGHT);
        est.setFont(new Font("Dialog", Font.ITALIC, 14));

        centru = new Label("Centru", Label.CENTER);
        centru.setBackground(Color.yellow);
        centru.setFont(new Font("Arial", Font.BOLD, 20));

        f.add(nord, BorderLayout.NORTH);
        f.add(sud, BorderLayout.SOUTH);
        f.add(est, BorderLayout.EAST);
        f.add(vest, BorderLayout.WEST);
    }
}
```

```
f.add(centru, BorderLayout.CENTER);

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

9.7.2 Clasa Button

Un obiect de tip `Button` reprezintă o componentă pentru plasarea unui buton etichetat pe o suprafața de afișare. Textul etichetei este format dintr-o singură linie.



Listing 9.21: Folosirea clasei `Button`

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

class Fereastră extends Frame implements ActionListener{

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

        setLayout(null);
        setSize(200, 120);

        Button b1 = new Button("OK");
        b1.setBounds(30, 30, 50, 70);
        b1.setFont(new Font("Arial", Font.BOLD, 14));
```

```
b1.setBackground(Color.orange);
add(b1);

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

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

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

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

9.7.3 Clasa Checkbox

Un obiect de tip `Checkbox` (comutator) reprezintă o componentă care se poate găsi în două stări: "selectată" sau "neselectată" (on/off). Acțiunea utilizatorului asupra unui comutator îl trece pe acesta în starea complementară celei în care se găsea. Este folosit pentru a prelua o anumită opțiune de la utilizator.



Listing 9.22: Folosirea clasei Checkbox

```

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

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

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

        setLayout(new GridLayout(5, 1));
        label1 = new Label("Ingrediente Pizza:", Label.CENTER);
        label1.setBackground(Color.orange);
        label2 = new Label("");
        label2.setBackground(Color.lightGray);

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

        add(label1);
        add(label2);
        add(cbx1);
        add(cbx2);
        add(cbx3);
    }
}

```

```
setSize(200, 200);

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

// Metoda interfetei ItemListener
public void itemStateChanged(ItemEvent e) {
    StringBuffer ingrediente = new StringBuffer();
    if (cbx1.getState() == true)
        ingrediente.append(" cascaval ");
    if (cbx2.getState() == true)
        ingrediente.append(" sunca ");
    if (cbx3.getState() == true)
        ingrediente.append(" ardei ");
    label2.setText(ingrediente.toString());
}
}

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

9.7.4 Clasa CheckboxGroup

Un obiect de tip `CheckboxGroup` definește un grup de comutatoare din care doar unul poate fi selectat. Uzual, aceste componente se mai numesc *butoane radio*. Această clasă nu este derivată din `Component`, oferind doar o modalitate de grupare a componentelor de tip `Checkbox`.



 Listing 9.23: Folosirea clasei CheckboxGroup

```

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

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

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

        setLayout(new GridLayout(5, 1));
        label1 = new Label("Alegeti postul TV", Label.CENTER);
        label1.setBackground(Color.orange);
        label2 = new Label("", Label.CENTER);
        label2.setBackground(Color.lightGray);

        cbg = new CheckboxGroup();
        cbx1 = new Checkbox("HBO", cbg, false);
        cbx2 = new Checkbox("Discovery", cbg, false);
        cbx3 = new Checkbox("MTV", cbg, false);

        add(label1);
        add(label2);
        add(cbx1);
  
```

```
add(cbx2);
add(cbx3);

setSize(200, 200);

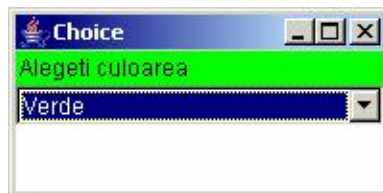
cbx1.addItemListener(this);
cbx2.addItemListener(this);
cbx3.addItemListener(this);
}

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

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

9.7.5 Clasa Choice

Un obiect de tip **Choice** definește o listă de opțiuni din care utilizatorul poate selecta una singură. La un moment dat, din întreaga listă doar o singură opțiune este vizibilă, cea selectată în momentul curent. O componentă **Choice** este însoțită de un buton etichetat cu o sgeată verticală la apăsarea căruia este afișată întreaga sa listă de elemente, pentru ca utilizatorul să poată selecta o anumită opțiune.



Listing 9.24: Folosirea clasei Choice

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

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

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

        setLayout(new GridLayout(4, 1));
        label = new Label("Alegeti culoarea");
        label.setBackground(Color.red);

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

        add(label);
        add(culori);

        setSize(200, 100);

        culori.addItemListener(this);
    }

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

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

9.7.6 Clasa List

Un obiect de tip `List` definește o listă de opțiuni care poate fi setată astfel încât utilizatorul să poată selecta o singură opțiune sau mai multe. Toate elementele listei sunt vizibile în limita dimensiunilor grafice ale componentei.



Listing 9.25: Folosirea clasei `List`

```
import java.awt.*;  
import java.awt.event.*;  
  
class Fereastra extends Frame implements ItemListener {  
    private Label label;  
    private List culori;  
  
    public Fereastra(String titlu) {  
        super(titlu);  
        this.addWindowListener(new WindowAdapter() {
```

```
        public void windowClosing(WindowEvent e) {
            System.exit(0);
        }
    });

    setLayout(new GridLayout(2, 1));
    label = new Label("Alegeti culoarea", Label.CENTER);
    label.setBackground(Color.red);

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

    add(label);
    add(culori);

    setSize(200, 200);

    culori.addItemListener(this);
}

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

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

9.7.7 Clasa ScrollBar

Un obiect de tip `Scrollbar` definește o bară de defilare pe verticală sau orizontală. Este utilă pentru punerea la dispoziția utilizatorului a unei modalități sugestive de a alege o anumită valoare dintr-un interval.



Listing 9.26: Folosirea clasei `Scrollbar`

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

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

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

        setLayout(new GridLayout(2, 1));

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

        scroll = new Scrollbar(Scrollbar.HORIZONTAL, 0, 1, 0,
            101);
        add(valoare);
        add(scroll);

        setSize(200, 80);

        scroll.addAdjustmentListener(this);
    }

    // Metoda interfetei AdjustmentListener
```

```

    public void adjustmentValueChanged(AdjustmentEvent e) {
        valoare.setText(scroll.getValue() + " %");
    }
}

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

```

9.7.8 Clasa ScrollPane

Un obiect de tip `ScrollPane` permite atașarea unor bare de defilare (orizontală și/sau verticală) oricărei componente grafice. Acest lucru este util pentru acele componente care nu au implementată funcționalitatea de defilare automată, cum ar fi listele (obiecte din clasa `List`).



Listing 9.27: Folosirea clasei `ScrollPane`

```

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

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

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

```

```
    }  
    });  
  
    list = new List(7);  
    list.add("Luni");  
    list.add("Marti");  
    list.add("Miercuri");  
    list.add("Joi");  
    list.add("Vineri");  
    list.add("Sambata");  
    list.add("Duminica");  
    list.select(1);  
  
    sp = new ScrollPane(ScrollPane.SCROLLBARS_ALWAYS);  
    sp.add(list);  
  
    add(sp, BorderLayout.CENTER);  
  
    setSize(200, 200);  
    }  
}  
  
public class TestScrollPane {  
    public static void main(String args[]) {  
        Fereastra f = new Fereastra("ScrollPane");  
        f.show();  
    }  
}
```

9.7.9 Clasa TextField

Un obiect de tip `TextField` definește un control de editare a textului pe o singură linie. Este util pentru interogarea utilizatorului asupra unor valori.



Listing 9.28: Folosirea clasei TextField

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

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

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

        setLayout(new GridLayout(3, 1));
        setBackground(Color.lightGray);

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

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

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

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

        setSize(350, 100);

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

```

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

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

```

9.7.10 Clasa TextArea

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



Listing 9.29: Folosirea clasei `TextArea`

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

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

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

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

        setBackground(Color.lightGray);
        text = new TextArea("", 30, 10, TextArea.
            SCROLLBARS_VERTICAL_ONLY);
        nume = new TextField("", 12);
        salvare = new Button("Salveaza text");
        salvare.setEnabled(false);

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

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

        setSize(300, 200);

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

    // Metoda interfetei TextListener
    public void textValueChanged(TextEvent e) {
        if (text.getText().length() == 0 ||
            nume.getText().length() == 0)
            salvare.setEnabled(false);
        else
            salvare.setEnabled(true);
    }
}
```

```
}

// Metoda interfetei ActionListener
public void actionPerformed(ActionEvent e) {
    String continut = text.getText();
    try {
        PrintWriter out = new PrintWriter(
            new FileWriter(ume.getText()));
        out.print(continut);
        out.close();
        text.requestFocus();
    } catch(IOException ex) {
        ex.printStackTrace();
    }
}

}

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

```

Capitolul 10

Desenarea

10.1 Conceptul de desenare

Un program Java care are interfață grafică cu utilizatorul trebuie să deseneze pe ecran toate componentele sale care au o reprezentare vizuală. Această desenare include componentele standard folosite în aplicație precum și cele definite de către programator. Desenarea componentelor se face automat și este un proces care se execută în următoarele situații:

- la afișarea pentru prima dată a unei componente;
- la operații de minimizare, maximizare, redimensionare a suprafeței de afișare;
- ca răspuns al unei solicitări explicite a programului.

Metodele care controlează procesul de desenare se găsesc în clasa **Component** și sunt următoarele:

- **void paint(Graphics g)** - Desenează o componentă. Este o metodă supradefinită de fiecare componentă în parte pentru a furniza reprezentarea sa grafică specifică. Metoda este apelată de fiecare dată când conținutul componentei trebuie desenat sau redesenat și nu va fi apelată explicit.
- **void update(Graphics g)** - Actualizează starea grafică a unei componente. Acțiunea acestei metode se realizează în trei pași:
 1. șterge componenta prin supradesenarea ei cu culoarea fundalului;

2. stabilește culoarea (foreground) a componentei;
 3. apelează metoda `paint` pentru a redesea componenta.
- **`void repaint()`** - Execută explicit un apel al metodei `update` pentru a actualiza reprezentarea grafică a unei componente.

După cum se observă, singurul argument al metodelor `paint` și `update` este un obiect de tip **Graphics**. Acesta reprezintă *contextul grafic* în care se execută desenarea componentelor (vezi "Contextul grafic de desenare - clasa Graphics").

Atenție

Toate desenele care trebuie să apară pe o suprafața de desenare se realizează în metoda `paint` a unei componente, în general apelată automat sau explicit cu metoda `repaint` ori de câte ori componenta respectivă trebuie redesenată. Există posibilitatea de a desena și în afara metodei `paint`, însă aceste desene se vor pierde la prima operație de minimizare, maximizare, redimensionare a suprafeței de afișare.

10.1.1 Metoda `paint`

După cum am spus, toate desenele care trebuie să apară pe o suprafață de afișare se realizează în metoda `paint` a unei componente. Metoda `paint` este definită în superclasa `Component` însă nu are nici o implementare și, din acest motiv, orice obiect grafic care dorește să se deseneze trebuie să o supradefinească pentru a-și crea propria sa reprezentare. Componentele standard AWT au deja supradefinită această metodă deci nu trebuie să ne preocupe desenarea lor, însă putem modifica reprezentarea lor grafică prin crearea unei subclase și supradefinirea metodei `paint`, având însă grijă să apelăm și metoda superclasei care se ocupă cu desenarea efectivă a componentei.

În exemplul de mai jos, redefinim metoda `paint` pentru un obiect de tip `Frame`, pentru a crea o clasă ce instanțiază ferestre pentru o aplicație demonstrativă (în colțul stânga sus este afișat textul "Aplicație DEMO").

Listing 10.1: Supradefinirea metodei `paint`

```
import java.awt.*;
class Fereastră extends Frame {
    public Fereastră(String titlu) {
        super(titlu);
        setSize(200, 100);
    }

    public void paint(Graphics g) {
        // Apelăm metoda paint a clasei Frame
        super.paint(g);
        g.setFont(new Font("Arial", Font.BOLD, 11));
        g.setColor(Color.red);
        g.drawString("Aplicatie DEMO", 5, 35);
    }
}

public class TestPaint {
    public static void main(String args[]) {
        Fereastră f = new Fereastră("Test paint");
        f.show();
    }
}
```

Observați că la orice redimensionare a ferestrei textul "Aplicatie DEMO" va fi redesenat. Dacă desenarea acestui text ar fi fost făcută oriunde în altă parte decât în metoda `paint`, la prima redimensionare a ferestrei acesta s-ar pierde.

Așadar, desenarea în Java trebuie să se facă doar în cadrul metodelor `paint` ale componentelor grafice.

10.1.2 Suprafețe de desenare - clasa `Canvas`

În afara posibilității de a utiliza componente grafice standard, Java oferă și posibilitatea controlului la nivel de punct (pixel) pe dispozitivul grafic, respectiv desenarea a diferite forme grafice direct pe suprafața unei componente. Deși este posibil, în general nu se desenează la nivel de pixel direct pe suprafața ferestrelor sau a altor containere, ci vor fi folosite clase dedicate acestui scop.

În AWT a fost definit un tip special de componentă numită **Canvas** (pânză de pictor), al cărei scop este de a fi extinsă pentru a implementa

obiecte grafice cu o anumită înfățișare. Așadar, `Canvas` este o clasă generică din care se derivează subclase pentru crearea suprafețelor de desenare (planșe). Planșele nu pot conține alte componente grafice, ele fiind utilizate doar ca suprafețe de desenat sau ca fundal pentru animație. Desenarea pe o planșă se face prin supradefinirea metodei `paint` a acesteia.

Concret, o *planșă* este o suprafață dreptunghiulară de culoare albă, pe care se poate desena. Dimensiunile sale implicite sunt 0 și, din acest motiv, este recomandat ca o planșă să redefinească metoda `getPreferredSize`, eventual și `getMinimumSize`, `getMaximumSize`, deoarece acestea vor fi apelate de către gestionarii de poziționare.

Etapele uzuale care trebuie parcurse pentru crearea unui desen, sau mai bine zis a unei componente cu o anumită înfățișare, sunt:

- crearea unei planșe de desenare, adică o subclasă a lui `Canvas`;
- redefinirea metodei `paint` din clasa respectivă;
- redefinirea metodelor `getPreferredSize`, eventual `getMinimumSize`, `getMaximumSize`;
- adăugarea planșei pe un container cu metoda `add`.
- tratarea evenimentelor de tip `FocusEvent`, `KeyEvent`, `MouseEvent`, `ComponentEvent`, dacă este cazul.

Definirea generică a unei planșe are următorul format:

```
class Plansa extends Canvas implements ...Listener {

    //Eventual, unul sau mai multi constructori
    public Plansa() {
        ...
    }
    // Metode de desenare a componentei
    public void paint(Graphics g) {
        ...
    }
    // Metodele folosite de gestionarii de pozitionare
    public Dimension getPreferredSize() {
        // Dimensiunea implicita a plansei
    }
}
```

```

    return ...;
}
public Dimension getMinimumSize() {
    return ...
}
public Dimension getMaximumSize() {
    return ...
}
// Implementarea metodelor interfetelor de tip Listener
...
}

```

Să definim o planșă pe care desenăm un pătrat și cercul său circumscris, colorate diferite. La fiecare click de mouse, vom interschimba cele două culori între ele.

Listing 10.2: Folosirea clasei Canvas

```

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

class Plansa extends Canvas {
    Dimension dim = new Dimension(100, 100);
    private Color color[] = {Color.red, Color.blue};
    private int index = 0;

    public Plansa() {
        this.addMouseListener(new MouseAdapter() {
            public void mouseClicked(MouseEvent e) {
                index = 1 - index;
                repaint();
            }
        });
    }

    public void paint(Graphics g) {
        g.setColor(color[index]);
        g.drawRect(0, 0, dim.width, dim.height);
        g.setColor(color[1 - index]);
        g.fillOval(0, 0, dim.width, dim.height);
    }

    public Dimension getPreferredSize() {

```

```
        return dim;
    }
}

class Fereastră extends Frame {
    public Fereastră(String titlu) {
        super(titlu);
        setSize(200, 200);
        add(new Plansa(), BorderLayout.CENTER);
    }
}

public class TestCanvas {
    public static void main(String args[]) {
        new Fereastră("Test Canvas").show();
    }
}
```

10.2 Contextul grafic de desenare

Înainte ca utilizatorul să poată desena, el trebuie să obțină un context grafic de desenare pentru suprafața căreia îi aparține regiunea pe care se va desena. Un *context grafic* este, de fapt, un obiect prin intermediul căruia putem controla procesul de desenare a unui obiect. În general, desenarea se poate face:

- pe o porțiune de ecran,
- la imprimantă sau
- într-o zonă virtuală de memorie.

Un context grafic este specificat prin intermediul unui obiect de tip **Graphics** primit ca parametru în metodele `paint` și `update`. În funcție de dispozitivul fizic pe care se face afișarea (ecran, imprimantă, plotter, etc) metodele de desenare au implementări interne diferite, transparente utilizatorului.

Clasa **Graphics** pune la dispoziție metode pentru:

- primitive grafice: desenarea de figuri geometrice, texte și imagini
- stabilirea proprietăților contextului grafic, adică stabilirea:

- culorii și fontului curente cu care se face desenarea,
- originii coordonatelor suprafeței de desenare,
- suprafeței în care sunt vizibile componentelor desenate,
- modului de desenare.

10.2.1 Proprietățile contextului grafic

La orice tip de desenare parametrii legați de culoare, font, etc. vor fi specificați pentru contextul grafic în care se face desenarea și nu vor fi trimiși ca argumente metodelor respective de desenare. În continuare, enumerăm aceste proprietăți și metodele asociate lor din clasa `Graphics`.

Proprietate	Metode
Culoarea de desenare	<code>Color getColor()</code> <code>void setColor(Color c)</code>
Fontul de scriere a textelor	<code>Font getFont()</code> <code>void setFont(Font f)</code>
Originea coordonatelor	<code>translate(int x, int y)</code>
Zona de decupare (zona în care sunt vizibile desenele)	<code>Shape getClip()</code> <code>void setClip(Shape s)</code>
Modul de desenare	<code>void setXorMode(Color c)</code> <code>void setPaintMode(Color c)</code>

10.2.2 Primitive grafice

Prin *primitive grafice* ne vom referi în continuare la metodele clasei `Graphics`, care permit desenarea de figuri geometrice și texte.

Desenarea textelor se face cu uzual cu metoda `drawString` care primește ca argumente un șir și colțul din stânga-jos al textului. Textul va fi desenat cu fontul și culoarea curente ale contextului grafic.

```
// Desenam la coordonatele x=10, y=20;
drawString("Hello", 10, 20);
```

Desenarea figurilor geometrice se realizează cu următoarele metode:

Figură geometrică	Metode
Linie	<code>drawLine</code> <code>drawPolyline</code>
Dreptunghi simplu	<code>drawRect</code> <code>fillRect</code> <code>clearRect</code>
Dreptunghi cu chenar "ridicat" sau "adâncit"	<code>draw3DRect</code> <code>fill3DRect</code>
Dreptunghi cu colțuri retunjite	<code>drawRoundRect</code> <code>fillRoundRect</code>
Poligon	<code>drawPolygon</code> <code>fillPolygon</code>
Oval (Elipsă)	<code>drawOval</code> <code>fillOval</code>
Arc circular sau eliptic	<code>drawArc</code> <code>fillArc</code>

Metodele care încep cu "fill" vor desena figuri geometrice care au interiorul colorat, adică "umplut" cu culoarea curentă a contextului de desenare, în timp ce metodele care încep cu "draw" vor desena doar conturul figurii respective.

10.3 Folosirea fonturilor

După cum vazut, pentru a scrie un text pe ecran avem două posibilități. Prima dintre acestea este să folosim o componentă orientată-text, cum ar fi `Label`, iar a doua să apelăm la metodele clasei `Graphics` de desenare a textelor, cum ar fi `drawString`. Indiferent de modalitatea aleasă, putem specifica prin intermediul fonturilor cum să arate textul respectiv, acest lucru realizându-se prin metoda `setFont` fie din clasa `Component`, fie din `Graphics`.

Cei mai importanți parametri ce caracterizează un font sunt:

- Numele fontului: Helvetica Bold, Arial Bold Italic, etc.
- Familia din care face parte fontul: Helvetica, Arial, etc.
- Dimensiunea fontului: înălțimea sa;
- Stilul fontului: **îngroșat (bold)**, *înclinat (italic)*;

- Metrica fontului.

Clasele care oferă suport pentru lucrul cu fonturi sunt **Font** și **FontMetrics**, în continuare fiind prezentate modalitățile de lucru cu acestea.

10.3.1 Clasa Font

Un obiect de tip **Font** încapsulează informații despre toți parametrii unui font, mai puțin despre metrica acestuia. Constructorul uzual al clasei este cel care primește ca argument numele fontului, dimensiunea și stilul acestuia:

```
Font(String name, int style, int size)
```

Stilul unui font este specificat prin intermediul constantelor: **Font.PLAIN**, **Font.BOLD**, **Font.ITALIC** iar dimensiunea printr-un întreg, ca în exemplele de mai jos:

```
new Font("Dialog", Font.PLAIN, 12);
new Font("Arial", Font.ITALIC, 14);
new Font("Courier", Font.BOLD, 10);
```

Folosirea unui obiect de tip **Font** se realizează uzual astfel:

```
// Pentru componente etichetate
Label label = new Label("Un text");
label.setFont(new Font("Dialog", Font.PLAIN, 12));

// In metoda paint(Graphics g)
g.setFont(new Font("Courier", Font.BOLD, 10));
g.drawString("Alt text", 10, 20);
```

O platformă de lucru are instalate, la un moment dat, o serie întreagă de fonturi care sunt disponibile pentru scrierea textelor. Lista acestor fonturi se poate obține astfel:

```
Font[] fonturi = GraphicsEnvironment.
    getLocalGraphicsEnvironment().getAllFonts();
```

Exemplul următor afișează lista tuturor fonturilor disponibile pe platforma curentă de lucru. Textul fiecărui nume de font va fi scris cu fontul său corespunzător.

Listing 10.3: Lucrul cu fonturi

```
import java.awt.*;

class Fonturi extends Canvas {
    private Font[] fonturi;
    Dimension canvasSize = new Dimension(400, 400);

    public Fonturi() {
        fonturi = GraphicsEnvironment.
            getLocalGraphicsEnvironment().getAllFonts();
        canvasSize.height = (1 + fonturi.length) * 20;
    }

    public void paint(Graphics g) {
        String nume;
        for(int i=0; i < fonturi.length; i++) {
            nume = fonturi[i].getFontName();
            g.setFont(new Font(nume, Font.PLAIN, 14));
            g.drawString(i + ". " + nume, 20, (i + 1) * 20);
        }
    }

    public Dimension getPreferredSize() {
        return canvasSize;
    }
}

class Fereastra extends Frame {
    public Fereastra(String titlu) {
        super(titlu);
        ScrollPane sp = new ScrollPane();
        sp.setSize(400, 400);
        sp.add(new Fonturi());
        add(sp, BorderLayout.CENTER);
        pack();
    }
}

public class TestAllFonts {
    public static void main(String args[]) {
        new Fereastra("All fonts").show();
    }
}
```

10.3.2 Clasa `FontMetrics`

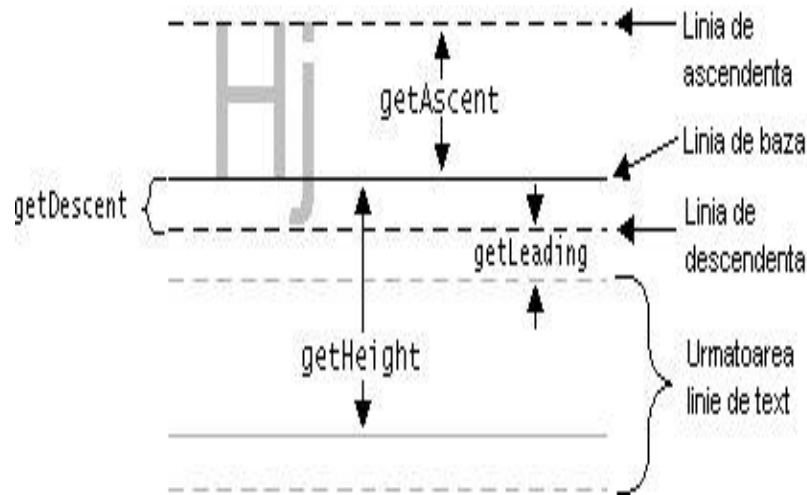
La afișarea unui șir cu metoda `drawString` trebuie să specificăm poziția la care să apară textul respectiv pe ecran. În momentul în care avem de afișat mai multe șiruri consecutiv, sau unele sub altele, trebuie să calculăm pozițiile lor de afișare în funcție de lungimea și înălțimea în pixeli a celorlalte texte. Pentru aceasta este folosită clasa **FontMetrics**. Un obiect din această clasă se construiește pornind de la un obiect de tip `Font` și pune la dispoziție informații despre dimensiunile în pixeli pe care le au caracterele fontului respectiv într-un anumit context de desenare.

Așadar, un obiect de tip `FontMetrics` încapsulează informații despre *metrica* unui font, cu alte cuvinte despre dimensiunile în pixeli ale caracterelor sale. Utilitatea principală a acestei clase constă în faptul că permite poziționarea precisă a textelor pe o suprafață de desenare, indiferent de fontul folosit de acestea.

Metrica unui font constă în următoarele atribute pe care le au caracterele sale:

- Linia de bază: este linia după care sunt aliniate caracterele unui font;
- Linia de ascendență: linia superioară pe care nu o depășeste nici un caracter din font
- Linia de descendență: linia inferioară sub care nu coboară nici un caracter din font;
- Ascendentul: distanța între linia de bază și linia de ascendență;
- Descendentul: distanța între linia de bază și linia de descendență;
- Lățimea: lățimea unui anumit caracter din font;
- Distanța între linii ("leading"): distanța optimă între două linii de text scrise cu același font.
- Înălțimea: distanța dintre liniile de bază ($\text{leading} + \text{ascent} + \text{descent}$);

Figura de mai jos prezintă o imagine reprezentativă asupra metricii unui font:



Reamintim că la metoda `drawString(String s, int x, int y)` argumentele x și y reprezintă colțul din **stânga-jos** al textului. Ca să fim mai preciși, y reprezintă poziția liniei de bază a textului care va fi scris.

Un context grafic pune la dispoziție o metodă specială `getFontMetrics` de creare a unui obiect de tip `FontMetrics`, pornind de la fontul curent al contextului grafic:

```
public void paint(Graphics g) {
    Font f = new Font("Arial", Font.BOLD, 11);
    FontMetrics fm = g.getFontMetrics();
}
```

Cele mai folosite metode ale clasei `FontMetrics` sunt:

- `getHeight` - determină înălțimea unei linii pe care vor fi scrise caractere ale unui font;
- `stringWidth` - determină lățimea totală în pixeli a unui șir de caractere specificat;
- `charWidth` - determină lățimea unui anumit caracter din font.

În exemplul următor sunt afișate pe ecran zilele săptămânii și lunile anului:

Listing 10.4: Folosirea clasei FontMetrics

```
import java.awt.*;

class Texte extends Canvas {
    Dimension canvasSize = new Dimension(800, 100);
    private String[] zile= {
        "Luni", "Marti", "Miercuri", "Joi",
        "Vineri", "Sambata", "Duminica"};
    private String[] luni = {
        "Ianuarie", "Februarie", "Martie", "Aprilie",
        "Mai", "Iunie", "Iulie", "August", "Septembrie",
        "Octombrie", "Noiembrie", "Decembrie"};

    public void paint(Graphics g) {
        FontMetrics fm;
        int x,y;
        String etZile = "Zilele saptamanii:",
            etLuni = "Lunile anului:", text;

        // Alegem un font si aflam metrica sa
        g.setFont(new Font("Arial", Font.BOLD, 20));
        fm = g.getFontMetrics();
        x = 0;
        y = fm.getHeight();
        g.drawString(etZile, x, y);
        x += fm.stringWidth(etZile);

        for(int i=0; i < zile.length; i++) {
            text = zile[i];
            if (i < zile.length - 1)
                text += ", ";
            g.drawString(text, x, y);
            x += fm.stringWidth(text);
        }
        // Schimbam fontul
        g.setFont(new Font("Dialog", Font.PLAIN, 14));
        fm = g.getFontMetrics();
        x = 0;
        y += fm.getHeight();
        g.drawString(etLuni, x, y);
        x += fm.stringWidth(etLuni);

        for(int i=0; i < luni.length; i++) {
            text = luni[i];
            if (i < luni.length - 1)
```

```

        text += ", ";
        g.drawString(text, x, y);
        x += fm.stringWidth(text);
    }
}
public Dimension getPreferredSize() {
    return canvasSize;
}
}

class Fereastra extends Frame {
    public Fereastra(String titlu) {
        super(titlu);
        add(new Texte(), BorderLayout.CENTER);
        pack();
    }
}

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

```

10.4 Folosirea culorilor

Orice culoare este formată prin combinația culorilor standard roșu (red), verde (green) și albastru (blue), la care se adaugă un anumit grad de transparență (alpha). Fiecare din acești patru parametri poate varia într-un interval cuprins fie între 0 și 255 (dacă dorim să specificăm valorile prin numere întregi), fie între 0.0 și 1.0 (dacă dorim să specificăm valorile prin numere reale).

O culoare este reprezentată printr-o instanță a clasei **Color** sau a sub-clasei sale **SystemColor**. Pentru a crea o culoare avem două posibilități:

- Să folosim una din constantele definite în cele două clase;
- Să folosim unul din constructorii clasei **Color**.

Să vedem mai întâi care sunt constantele definite în aceste clase:

Color	SystemColor
black	activeCaption
blue	activeCaptionBorder
cyan	activeCaptionText
darkGray	control
gray	controlHighlight
green	controlShadow
lightGray	controlText
magenta	desktop
orange	menu
pink	text
red	textHighlight
white	window
yellow	...

Observați că în clasa `Color` sunt definite culori uzuale din paleta standard de culori, în timp ce în clasa `SystemColor` sunt definite culorile componentelor standard (ferestre, texte, meniuri, etc) ale platformei curente de lucru. Folosirea acestor constante se face ca în exemplele de mai jos:

```
Color rosu = Color.red;
Color galben = Color.yellow;
Color fundal = SystemColor.desktop;
```

Dacă nici una din aceste culori predefinite nu corespunde preferințelor noastre, atunci putem crea noi culori prin intermediul constructorilor clasei `Color`:

```
Color(float red, float green, float blue)
Color(float red, float green, float blue, float alpha)
Color(int red, int green, int blue)
Color(int red, int green, int blue, int alpha)
Color(int rgb)
```

unde *red*, *green*, *blue*, *alpha* sunt valorile pentru roșu, verde, albastru și transparență iar parametrul "rgb" de la ultimul constructor reprezintă un întreg format din: biții 16-23 roșu, 8-15 verde, 0-7 albastru.

Valorile argumentelor variază între 0 – 255 pentru tipul `int`, respectiv 0.0 – 1.0 pentru tipul `float`. Valoarea 255 (sau 1.0) pentru transparență specifică faptul că respectiva culoare este complet opacă, iar valoarea 0 (sau 0.0) specifică transparență totală. Implicit, culorile sunt complet opace.

```
// Exemple de folosire a constructorilor:
Color alb = new Color(255, 255, 255);
Color negru = new Color(0, 0, 0);
Color rosu = new Color(255, 0, 0);
Color rosuTransparent = new Color(255, 0, 0, 128);
```

Metodele cele mai folosite ale clasei Color sunt:

brighter darker	Creează o nouă versiune a culorii curente mai deschisă, respectiv mai închisă
getRed getGreen getBlue getAlpha	Determină parametrii din care este alcătuită culoarea
getRGB	Determină valoarea ce reprezintă culoarea respectivă (biții 16-23 roșu, 8-15 verde, 0-7 albastru)

Să considerăm o aplicație cu ajutorul căreia putem vizualiza dinamic culorile obținute prin diferite combinații ale parametrilor ce formează o culoare. Aplicația va arăta astfel:



Listing 10.5: Folosirea clasei Color

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

class Culoare extends Canvas {
    public Color color = new Color(0, 0, 0, 255);
    Dimension canvasSize = new Dimension(150, 50);

    public void paint(Graphics g) {
        g.setColor(Color.black);
```

```

    g.setFont(new Font("Arial", Font.BOLD, 12));
    String text = "";
    text += " R=" + color.getRed();
    text += " G=" + color.getGreen();
    text += " B=" + color.getBlue();
    text += " A=" + color.getAlpha();
    g.drawString(text, 0, 30);

    g.setColor(color);
    g.fillRect(0, 0, canvasSize.width, canvasSize.height);
}

public Dimension getPreferredSize() {
    return canvasSize;
}
}

class Fereastra extends Frame implements AdjustmentListener {
    private Scrollbar rValue, gValue, bValue, aValue;
    private Culoare culoare;

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

        Panel rgbValues = new Panel();
        rgbValues.setLayout(new GridLayout(4, 1));
        rValue = new Scrollbar(Scrollbar.HORIZONTAL, 0, 1, 0,
            256);
        rValue.setBackground(Color.red);

        gValue = new Scrollbar(Scrollbar.HORIZONTAL, 0, 1, 0,
            256);
        gValue.setBackground(Color.green);

        bValue = new Scrollbar(Scrollbar.HORIZONTAL, 0, 1, 0,
            256);
        bValue.setBackground(Color.blue);

        aValue = new Scrollbar(Scrollbar.HORIZONTAL, 0, 1, 0,
            256);

```

```
aValue.setValue(255);
aValue.setBackground(Color.lightGray);

rgbValues.add(rValue);
rgbValues.add(gValue);
rgbValues.add(bValue);
rgbValues.add(aValue);
rgbValues.setSize(200, 100);
add(rgbValues, BorderLayout.CENTER);

culoare = new Culoare();
add(culoare, BorderLayout.NORTH);

pack();

rValue.addAdjustmentListener(this);
gValue.addAdjustmentListener(this);
bValue.addAdjustmentListener(this);
aValue.addAdjustmentListener(this);
}

public void adjustmentValueChanged(AdjustmentEvent e) {
    int r = rValue.getValue();
    int g = gValue.getValue();
    int b = bValue.getValue();
    int a = aValue.getValue();
    Color c = new Color(r, g, b, a);
    culoare.color = c;
    culoare.repaint();
}
}

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

10.5 Folosirea imaginilor

Aceasta este o imagine:



În AWT este posibilă folosirea imaginilor create extern în format *gif* sau *jpeg*. Orice imagine va fi reprezentată ca o instanță a clasei **Image**. Aceasta nu este o clasă de componente (nu extinde **Component**) ci implementează obiecte care pot fi desenate pe suprafața unor componente cu metode specifice unui context grafic pentru componenta respectivă (similar modului cum se desenează o linie sau un cerc).

10.5.1 Afișarea imaginilor

Afișarea unei imagini presupune realizarea următoarelor doi pași:

1. Crearea unui obiect de tip **Image**;
2. Afișarea propriu-zisă într-un context grafic;

Crearea unui obiect de tip **Image** se face folosind o imagine dintr-un fișier fie aflat pe mașina pe care se lucrează, fie aflat la o anumită adresă Web (URL). Metodele pentru încărcarea unei imagini dintr-un fișier se găsesc în clasele **Applet** și **Toolkit**, având însă aceeași denumire **getImage** și următoarele formate:

Applet	Toolkit
<code>getImage(URL url)</code>	<code>getImage(URL url)</code>
<code>getImage(URL url, String fisier)</code>	<code>getImage(String fisier)</code>

Pentru a obține un obiect de tip **Toolkit** se va folosi metoda `getDefaultToolkit`, ca în exemplul de mai jos:

```
Toolkit toolkit = Toolkit.getDefaultToolkit();
Image image1 = toolkit.getImage("poza.gif");
Image image2 = toolkit.getImage(
    new URL("http://www.infoiasi.ro/~acf/poza.gif"));
```


Metoda `getImage` nu verifică dacă fișierul sau adresa specificată reprezintă o imagine validă și nici nu încarcă efectiv imaginea în memorie, aceste operațiuni fiind făcute abia în momentul în care se va realiza afișarea imaginii pentru prima dată. Metoda nu face decât să creeze un obiect de tip `Image` care face referință la o anumită imagine externă.

Afișarea unei imagini într-un context grafic se realizează prin intermediul metodei `drawImage` din clasa `Graphics` și, în general, va fi făcută în metoda `paint` a unei componente. Cele mai uzuale formate ale metodei sunt:

```
boolean drawImage(Image img, int x, int y, ImageObserver observer)
boolean drawImage(Image img, int x, int y, Color bgcolor,
    ImageObserver observer)
boolean drawImage(Image img, int x, int y, int width, int height,
    ImageObserver observer)
boolean drawImage(Image img, int x, int y, int width, int height,
    Color bgcolor, ImageObserver observer)
```

unde:

- *img* este obiectul ce reprezintă imaginea;
- *x, y* sunt coordonatele stânga-sus la care va fi afișată imaginea, relative la spațiul de coordonate al contextului grafic;
- *observer* este un obiect care "observă" încărcarea imaginii și va fi informat pe măsura derulării acesteia;
- *width, height* reprezintă înălțimea și lățimea la care trebuie scalată imaginea (dacă lipsesc, imaginea va fi afișată la dimensiunile ei reale);
- *bgColor* reprezintă culoarea cu care vor fi colorați pixelii transparenți ai imaginii (poate să lipsească).

În exemplul următor afișăm aceeași imagine de trei ori, folosind forme diferite ale metodei `drawImage`:

```
Image img = Toolkit.getDefaultToolkit().getImage("taz.gif");
g.drawImage(img, 0, 0, this);
g.drawImage(img, 0, 200, 100, 100, this);
g.drawImage(img, 200, 0, 200, 400, Color.yellow, this);
```

Metoda `drawImage` returnează `true` dacă imaginea a fost afișată în întregime și `false` în caz contrar, cu alte cuvinte metoda nu așteaptă ca o imagine să fie complet afișată ci se termină imediat ce procesul de afișare a început. În secțiunea următoare vom detalia acest aspect.

10.5.2 Monitorizarea încărcării imaginilor

În cazul în care se afișează o imagine care se găsește pe Internet sau imaginea afișată este de dimensiuni mari se va observa că aceasta nu apare complet de la început ci este desenată treptat, fără intervenția programatorului. Acest lucru se întâmplă deoarece metoda `drawImage` nu face decât să declanșeze procesul de încărcare și desenare a imaginii, după care redă imediat controlul apelantului, lucru deosebit de util întrucât procesul de încărcare a unei imagini poate dura mult și nu este de dorit ca în acest interval de timp (până la încărcarea completă a imaginii) aplicația să fie blocată. Ca urmare, la apelul metodei `drawImage` va fi desenată numai porțiunea de imagine care este disponibilă la momentul inițial și care poate fi incompletă. De aceea trebuie să existe un mecanism prin care componenta să fie redesenată automat în momentul în care au mai sosit informații legate de imagine, până la afișarea sa completă. Acest mecanism este realizat prin intermediul interfeței **ImageObserver**, implementată de clasa **Component** și deci de toate componentele. Această interfață descrie obiecte care au început să utilizeze o imagine incompletă și care trebuie anunțate de noile date obținute în legătură cu imaginea respectivă.

Interfața **ImageObserver** are o singură metodă numită **imageUpdate**, ce va fi apelată periodic de firul de execuție (creat automat) care se ocupă cu încărcarea imaginii. Formatul acestei metode este:

```
boolean imageUpdate (Image img, int flags,  
                    int x, int y, int w, int h )
```

Implementarea implicită constă dintr-un apel la metoda `repaint` pentru dreptunghiul specificat la apel și care reprezintă zona din imagine pentru care se cunosc noi informații. Intregul *flags* furnizează informații despre starea transferului. Aceste informații pot fi aflate prin intermediul constantelor definite de interfață:

ABORT	Încărcarea imaginii a fost întreruptă, înainte de completarea sa
ALLBITS	Imaginea a fost încărcată complet
ERROR	A apărut o eroare în timpul încărcării imaginii
FRAMEBITS	Toți biții cadrului curent sunt disponibili
HEIGHT	Înălțimea imaginii este disponibilă
PROPERTIES	Proprietățile imaginii sunt disponibile
SOMEBITS	Au fost recepționați noi pixeli ai imaginii
WIDTH	Lățimea imaginii este disponibilă

Prezența în parametrul *flags* a unui bit de valoare 1 pe poziția reprezentată de o constantă înseamnă că respectiva condiție este îndeplinită.

```
// Imaginea este completa
(flags & ALLBITS) != 0

// Eroare sau transferul imaginii a fost intrerupt
(flags & ERROR | ABORT ) != 0
```

Metoda `imageUpdate` poate fi redefinită de o componentă pentru a personaliza procesul de afișare al imaginii. Aceasta va fi apelată apelată asincron de fiecare dată când sunt disponibili noi pixeli.

```
public boolean imageUpdate(Image img, int flags,
    int x, int y, int w, int h) {
    // Desenam imaginea numai daca toti bitii sunt disponibili
    if (( flags & ALLBITS) != 0)
        repaint();

    // Daca sunt toti bitii nu mai sunt necesare noi update-uri
    return ( (flags & (ALLBITS | ABORT)) == 0);
}
```

De asemenea, se observă că metodele clasei `Image` pentru determinarea dimensiunilor unei imagini au ca argument un obiect de tip `ImageObserver`.

```
int getHeight(ImageObserver observer)
int getWidth(ImageObserver observer)
```

Dacă desenarea se face folosind clasa `Canvas`, atunci argumentul *observer* al metodelor referitoare la imagini va fi **this**.

10.5.3 Mecanismul de "double-buffering"

Tehnica de double-buffering implică realizarea unui desen în memorie și apoi transferul său pe ecran, pentru a elimina efectul neplăcut de "clipire" ("flickering") rezultat atunci când sunt efectuate redesenări repetate la intervale mici de timp. O situație frecventă în care se apelează la double-buffering este crearea de animații.

Secvența generală de implementare a mecanismului de double-buffering este următoarea:

```
// Supradefinim update pentru a elimina stergerea desenului
public void update(Graphics g) {
    paint(g);
}

public void paint(Graphics g) {
    // Desenam in memorie pe un obiect de tip Image
    // w si h sunt dimensiunile desenului
    Image img = createImage(w, h);
    Graphics gmem = img.getGraphics();

    /* Realizam desenul folosind gmem
       gmem.setColor(...);
       gmem.fillOval(...);
       ...
    */

    // Transferam desenul din memorie pe ecran
    // desenand de fapt imaginea creata
    g.drawImage(img, 0, 0, this);

    gmem.dispose();
}
}
```

10.5.4 Salvarea desenelor în format JPEG

Pachetul `com.sun.image.codec.jpeg` din distribuția standard Java oferă suport pentru salvarea unei imagini aflate în memorie într-un fișier în for-

mat JPEG. O clasă responsabilă cu realizarea acestei operațiuni ar putea fi definită astfel:

```
import com.sun.image.codec.jpeg.*;
import java.awt.image.BufferedImage;
import java.awt.*;
import java.io.*;

class JPEGWriter {

    static float quality = 0.9f; //intre 0 si 1
    public static void write(BufferedImage img, String filename) {

        try {
            FileOutputStream out = new FileOutputStream(filename);

            JPEGImageEncoder encoder =
                JPEGCodec.createJPEGEncoder(out);
            JPEGEncodeParam jep =
                encoder.getDefaultJPEGEncodeParam(img);
            jep.setQuality(quality, false);

            // Folosim setarile de codare jpeg implicite
            encoder.setJPEGEncodeParam(jep);
            encoder.encode(img);

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

10.5.5 Crearea imaginilor în memorie

În cazul în care dorim să folosim o anumită imagine creată direct din program și nu încărcată dintr-un fișier vom folosi clasa **MemoryImageSource**, aflată în pachetul `java.awt.image`. Pentru aceasta va trebui să definim un

vector de numere întregi în care vom scrie valorile întregi (RGB) ale culorilor pixelilor ce definesc imaginea noastră. Dimensiunea vectorului va fi înălțimea înmulțită cu lățimea în pixeli a imaginii. Constructorul clasei `MemoryImageSource` este:

```
MemoryImageSource(int w, int h, int[] pixeli, int off, int scan)
```

unde:

- w, h reprezintă dimensiunile imaginii (lățimea și înălțimea);
- `pixeli[]` este vectorul cu culorile imaginii;
- $off, scan$ reprezintă modalitatea de construire a matricii imaginii pornind de la vectorul cu pixeli, normal aceste valori sunt $off = 0, scan = w$

În exemplul următor vom crea o imagine cu pixeli de culori aleatorii și o vom afișa pe ecran:

```
int w = 100;
int h = 100;
int[] pix = new int[w * h];
int index = 0;
for (int y = 0; y < h; y++) {
for (int x = 0; x < w; x++) {
    int red = (int) (Math.random() * 255);
    int green = (int) (Math.random() * 255);
    int blue = (int) (Math.random() * 255);
    pix[index++] = new Color(red, green, blue).getRGB();
}

img = createImage(new MemoryImageSource(w, h, pix, 0, w));
g.drawImage(img, 0, 0, this); // g este un context grafic
```

10.6 Tipărirea

Tipărirea în Java este tratată în aceeași manieră ca și desenarea, singurul lucru diferit fiind contextul grafic în care se execută operațiile. Pachetul care oferă suport pentru tipărire este `java.awt.print`, iar clasa principală care controlează procesul de tipărire este `PrinterJob`. O aplicație va apela metode ale acestei clase pentru:

- Crearea unei sesiuni de tipărire (job);
- Invocarea dialogului cu utilizatorul pentru specificarea unor parametri legați de tipărire;
- Tipărirea efectivă.

Orice componentă care poate fi afișată pe ecran poate fi și tipărită. În general, orice informații care trebuie atât afișate cât și tipărite, vor fi încapsulate într-un obiect grafic - componentă, care are o reprezentare vizuală descrisă de metoda `paint` și care va specifica și modalitatea de reprezentare a sa la imprimantă.

Un obiect care va fi tipărit trebuie să implementeze interfața **Printable**, care conține o singură metodă **print**, responsabilă cu descrierea modalității de tipărire a obiectului. În cazul când imaginea de pe ecran coincide cu imaginea de la imprimantă, metodele `paint` și `print` pot specifica aceeași secvență de cod. În general, metoda `print` are următorul format:

```
public int print(Graphics g, PageFormat pf, int pageIndex)
throws PrinterException {

    // Descrierea imaginii obiectului ce va fi afisata la imprimanta
    // Poate fi un apel la metoda paint: paint(g)

    if (ceva nu este in regula) {
        return Printable.NO_SUCH_PAGE;
    }

    return Printable.PAGE_EXISTS;
}
```

Pașii care trebuie efectuați pentru tipărirea unui obiect sunt:

1. Crearea unei sesiuni de tipărire: `PrinterJob.getPrinterJob`
2. Specificarea obiectului care va fi tipărit: `setPrintable`; acesta trebuie să implementeze interfața **Printable**;
3. Opțional, inițierea unui dialog cu utilizatorul pentru precizarea unor parametri legați de tipărire: `printDialog`;

4. Tipărirea efectivă: `print`.

În exemplul următor vom defini un obiect care are aceeași reprezentare pe ecran cât și la imprimantă (un cerc circumscris unui pătrat, însoțite de un text) și vom tipări obiectul respectiv.

Listing 10.6: Tipărirea unei componente

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

class Plansa extends Canvas implements Printable {
    Dimension d = new Dimension(400, 400);
    public Dimension getPreferredSize() {
        return d;
    }

    public void paint(Graphics g) {
        g.drawRect(200, 200, 100, 100);
        g.drawOval(200, 200, 100, 100);
        g.drawString("Hello", 200, 200);
    }

    public int print(Graphics g, PageFormat pf, int pi)
        throws PrinterException {
        if (pi >= 1)
            return Printable.NO_SUCH_PAGE;

        paint(g);
        g.drawString("Numai la imprimanta", 200, 300);

        return Printable.PAGE_EXISTS;
    }
}

class Fereastra extends Frame implements ActionListener {
    private Plansa plansa = new Plansa();
    private Button print = new Button("Print");

    public Fereastra(String titlu) {
        super(titlu);
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
```



```
        System.exit(0);
    }
});

add(plansa, BorderLayout.CENTER);

Panel south = new Panel();
south.setLayout(new FlowLayout(FlowLayout.CENTER));
south.add(print);
add(south, BorderLayout.SOUTH);

print.addActionListener(this);
pack();
}

public void actionPerformed(ActionEvent e) {
    // 1. Crearea unei sesiuni de tiparire
    PrinterJob printJob = PrinterJob.getPrinterJob();

    // 2. Stabilirea obiectului ce va fi tiparit
    printJob.setPrintable(plansa);

    // 3. Initierea dialogului cu utilizatorul
    if (printJob.printDialog()) {
        try {
            // 4. Tiparirea efectiva
            printJob.print();
        } catch (PrinterException ex) {
            System.out.println("Exceptie la tiparire!");
            ex.printStackTrace();
        }
    }
}

}

public class TestPrint {
    public static void main(String args[]) throws Exception {
        Fereastra f = new Fereastra("Test Print");
        f.show();
    }
}
```

Tipărirea textelor

O altă variantă pentru tipărirea de texte este deschiderea unui flux către dispozitivul special reprezentat de imprimantă și scrierea informațiilor, linie cu linie, pe acest flux. În sistemul de operare Windows, imprimanta poate fi referită prin **"lpt1"**, iar în Unix prin **"/dev/lp"**. Observați că această abordare nu este portabilă, deoarece necesită tratare specială în funcție de sistemul de operare folosit.

Listing 10.7: Tipărirea textelor

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

class TestPrintText {
    public static void main(String args[]) throws Exception {
        // pentru Windows
        PrintWriter imp = new PrintWriter(new FileWriter("lpt1"))
            ;

        // pentru UNIX
        //PrintWriter imp = new PrintWriter(new FileWriter("/dev/
        lp"));

        imp.println("Test imprimanta");
        imp.println("ABCDE");
        imp.close();
    }
}
```

Capitolul 11

Swing

11.1 Introducere

11.1.1 JFC

Tehnologia Swing face parte dintr-un proiect mai amplu numit **JFC** (Java Foundation Classes) care pune la dispoziție o serie întreagă de facilități pentru scrierea de aplicații cu o interfață grafică mult îmbogățită funcțional și estetic față de vechiul model AWT. În JFC sunt incluse următoarele:

- **Componente Swing**

Sunt componente ce înlocuiesc și în același timp extind vechiul set oferit de modelul AWT.

- **Look-and-Feel**

Permite schimbarea înfățișării și a modului de interacțiune cu aplicația în funcție de preferințele fiecăruia. Același program poate utiliza diverse moduri Look-and-Feel, cum ar fi cele standard Windows, Mac, Java, Motif sau altele oferite de diverși dezvoltatori, acestea putând fi interschimbate de către utilizator chiar la momentul execuției .

- **Accessibility API**

Permite dezvoltarea de aplicații care să comunice cu dispozitive utilizate de către persoane cu diverse tipuri de handicap, cum ar fi cititoare de ecran, dispozitive de recunoaștere a vocii, ecrane Braille, etc.

- **Java 2D API**

Folosind Java 2D pot fi create aplicații care utilizează grafică la un

nivel avansat. Clasele puse la dispoziție permit crearea de desene complexe, efectuarea de operații geometrice (rotiri, scalări, translații, etc.), prelucrarea de imagini, tipărire, etc.

- **Drag-and-Drop**

Oferă posibilitatea de a efectua operații drag-and-drop între aplicații Java și aplicații native.

- **Internaționalizare**

Internaționalizarea și localizarea aplicațiilor sunt două facilități extrem de importante care permit dezvoltarea de aplicații care să poată fi configurate pentru exploatarea lor în diverse zone ale globului, utilizând limba și particularitățile legate de formatarea datei, numerelor sau a monedei din zona respectivă.

În acest capitol vom face o prezentare scurtă a componentelor Swing, deoarece prezentarea detaliată a tuturor facilităților oferite de JFC ar oferi suficient material pentru un volum de sine stătător.

11.1.2 Swing API

Unul din principalele deziderate ale tehnologiei Swing a fost să pună la dispoziție un set de componente GUI extensibile care să permită dezvoltarea rapidă de aplicații Java cu interfață grafică competitivă din punct de vedere comercial. Pentru a realiza acest lucru, API-ul oferit de Swing este deosebit de complex având 17 pachete în care se găsesc sute de clase și interfețe. Lista completă a pachetelor din distribuția standard 1.4 este dată în tabelul de mai jos:

<code>javax.accessibility</code>	<code>javax.swing.plaf</code>
<code>javax.swing.text.html</code>	<code>javax.swing</code>
<code>javax.swing.plaf.basic</code>	<code>javax.swing.text.parser</code>
<code>javax.swing.border</code>	<code>javax.swing.plaf.metal</code>
<code>javax.swing.text.rtf</code>	<code>javax.swing.colorchooser</code>
<code>javax.swing.plaf.multi</code>	<code>javax.swing.tree</code>
<code>javax.swing.event</code>	<code>javax.swing.table</code>
<code>javax.swing.undo</code>	<code>javax.swing.filechooser</code>
<code>javax.swing.text</code>	

Evident, nu toate aceste pachete sunt necesare la dezvoltarea unei aplicații, cel mai important și care conține componentele de bază fiind **javax.swing**.

Componentele folosite pentru crearea interfețelor grafice Swing pot fi grupate astfel:

- **Componente atomice**
JLabel, JButton, JCheckBox, JRadioButton, JToggleButton, JScrollBar, JSlider, JProgressBar, JSeparator
- **Componente complexe**
JTable, JTree, JComboBox, JSpinner, JList, JFileChooser, JColorChooser, JOptionPane
- **Componente pentru editare de text**
JTextField, JFormattedTextField, JPasswordField, JTextArea, JEditorPane, JTextPane
- **Meniuri**
JMenuBar, JMenu, JPopupMenu, JMenuItem, JCheckboxMenuItem, JRadioButtonMenuItem
- **Containere intermediare**
JPanel, JScrollPane, JSplitPane, JTabbedPane, JDesktopPane, JToolBar
- **Containere de nivel înalt**
JFrame, JDialog, JWindow, JInternalFrame, JApplet

11.1.3 Asemănări și deosebiri cu AWT

Nu se poate spune că Swing înlocuiește modelul AWT ci îl extinde pe acesta din urmă adăugându-i noi componente care fie înlocuiesc unele vechi fie sunt cu totul noi. O convenție în general respectată este prefixarea numelui unei clase AWT cu litera "J" pentru a denumi clasa corespondentă din Swing. Astfel, în locul clasei `java.awt.Button` putem folosi `javax.swing.JButton`, în loc de `java.awt.Label` putem folosi `javax.swing.JLabel`, etc. Este recomandat ca o aplicație cu interfață grafică să folosească fie componente AWT, fie Swing, amestecarea lor fiind mai puțin uzuală.

Aplicațiile GUI vor avea în continuare nevoie de pachetul `java.awt` deoarece aici sunt definite unele clase utilitare cum ar fi `Color`, `Font`, `Dimension`, etc. care nu au fost rescrise în Swing.

De asemenea, pachetul `java.awt.event` rămâne în continuare esențial pentru tratarea evenimentelor generate atât de componente AWT cât și de cele din Swing. Pe lângă acesta mai poate fi necesar și `javax.swing.event` care descrie tipuri de evenimente specifice unor componente Swing, mecanismul de tratare a lor fiind însă același ca în AWT.

Poziționarea componentelor este preluată din AWT, fiind adăugate însă noi clase care descriu gestionari de poziționare în completarea celor existente, cum ar fi `BoxLayout` și `SpringLayout`. Diferă însă modul de lucru cu containere, după cum vom vedea în secțiunea dedicată acestora.

Majoritatea componentelor Swing care permit afișarea unui text ca parte a reprezentării lor GUI pot specifica acel text fie în mod normal folosind un anumit font și o anumită culoare ce pot fi setate cu metodele `setFont` și `setColor`, fie prin intermediul limbajului HTML. Folosirea HTML aduce o flexibilitate deosebită în realizarea interfeței grafice, întrucât putem aplica formătări multiple unui text, descompunerea acestuia pe mai multe linii, etc., singurul dezavantaj fiind încetinirea etapei de afișare a componentelor.

```

JButton simplu = new JButton("Text simplu");
JButton html = new JButton(
    "<html><u>Text</u> <i>formatat</i></html>");

```

Să descriem o aplicație simplă folosind AWT și apoi Swing, pentru a ne crea o primă impresie asupra diferențelor și asemănărilor dintre cele două modele.

Listing 11.1: O aplicație simplă AWT

```

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

public class ExempluAWT extends Frame implements
    ActionListener {
    public ExempluAWT(String titlu) {
        super(titlu);
        setLayout(new FlowLayout());
        add(new Label("Hello AWT"));
        Button b = new Button("Close");
        b.addActionListener(this);
    }
}

```

```

        add(b);
        pack();
        show();
    }
    public void actionPerformed(ActionEvent e) {
        System.exit(0);
    }
    public static void main(String args[]) {
        new ExempluAWT("Hello");
    }
}

```

Listing 11.2: Aplicația rescrisă folosind Swing

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class ExempluSwing extends JFrame implements
    ActionListener {
    public ExempluSwing(String titlu) {
        super(titlu);
        // Metoda setLayout nu se aplica direct ferestrei
        getContentPane().setLayout(new FlowLayout());

        // Componentele au denumiri ce incep cu litera J
        // Textul poate fi si in format HTML
        getContentPane().add(new JLabel(
            "<html><u>Hello</u> <i>Swing</i></html>"));
        JButton b = new JButton("Close");
        b.addActionListener(this);

        // Metoda add nu se aplica direct ferestrei
        getContentPane().add(b);
        pack();
        show();
    }
    public void actionPerformed(ActionEvent e) {
        // Tratarea evenimentelor se face ca in AWT
        System.exit(0);
    }
    public static void main(String args[]) {
        new ExempluSwing("Hello");
    }
}

```

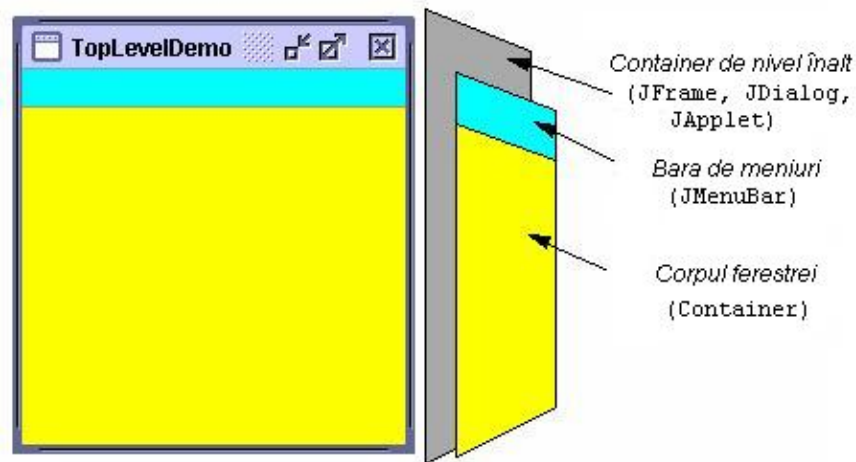

}

11.2 Folosirea ferestrelor

Pentru a fi afișate pe ecran componentele grafice ale unei aplicații trebuie plasate pe o suprafață de afișare (container). Fiecare componentă poate fi conținută doar într-un singur container, adăugarea ei pe o suprafață nouă de afișare determinând eliminarea ei de pe vechiul container pe care fusese plasată. Întrucât containerele pot fi încapsulate în alte containere, o componentă va face parte la un moment dat dintr-o ierarhie. Rădăcina acestei ierarhii trebuie să fie un așa numit *container de nivel înalt*, care este reprezentat de una din clasele **JFrame**, **JDialog** sau **JApplet**. Întrucât de appleturi ne vom ocupa separat, vom analiza în continuare primele două clase.

În general orice aplicație Java independentă bazată pe Swing conține cel puțin un container de nivel înalt reprezentat de fereastra principală a programului, instanță a clasei **JFrame**.

Simplificat, un obiect care reprezintă o fereastră Swing conține o zonă care este rezervată barei de meniuri și care este situată de obicei în partea sa superioară și corpul ferestrei pe care vor fi plasate componentele. Imaginea de mai jos pune în evidență această separare, valabilă de altfel pentru orice container de nivel înalt:



Corpul ferestrei este o instanță a clasei **Container** ce poate fi obținută cu metoda **getContentPane**. Plasarea și aranjarea componentelor pe suprafața

ferestrei se va face deci folosind obiectul de tip `Container` și nu direct fereastra. Așadar, deși este derivată din `Frame`, clasa `JFrame` este folosită într-un mod diferit față de părintele său:

```
Frame f = new Frame();
f.setLayout(new FlowLayout());
f.add(new Button("OK"));
```

```
JFrame jf = new JFrame();
jf.getContentPane().setLayout(new FlowLayout());
jf.getContentPane().add(new JButton("OK"));
```

Spre deosebire de `Frame`, un obiect `JFrame` are un comportament implicit la închiderea ferestrei care constă în ascunderea ferestrei atunci când utilizatorul apasă butonul de închidere. Acest comportament poate fi modificat prin apelarea metodei `setDefaultCloseOperation` care primește ca argument diverse constante ce se găsesc fie în clasa `WindowConstants`, fie chiar în `JFrame`.

```
jf.setDefaultCloseOperation(WindowConstants.HIDE_ON_CLOSE);
jf.setDefaultCloseOperation(WindowConstants.DO_NOTHING_ON_CLOSE);
jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Adăugarea unei bare de meniuri se realizează cu metoda `setJMenuBar`, care primește o instanță de tip `JMenuBar`. Crearea meniurilor este similară cu modelul AWT.

11.2.1 Ferestre interne

Din punctul de vedere al folosirii ferestrelor, aplicațiile pot fi împărțite în două categorii:

- *SDI (Single Document Interface)*
- *MDI (Multiple Document Interface)*

Programele din prima categorie gestionează la un moment dat o singură fereastră în care se găsesc componentele cu care interacționează utilizatorul. În a doua categorie, fereastra principală a aplicației înglobează la rândul ei alte ferestre, uzual cu funcționalități similare, ce permit lucrul concurent pe mai multe planuri.

În Swing, clasa **JInternalFrame** pune la dispoziție o modalitate de a crea ferestre în cadrul altor ferestre. Ferestrele interne au aproximativ aceeași înfățișare și funcționalitate cu ferestrele de tip **JFrame**, singura diferență fiind modul de gestionare a acestora.

Uzual, obiectele de tip **JInternalFrame** vor fi plasate pe un container de tip **DesktopPane**, care va fi apoi plasat pe o fereastră de tip **JFrame**. Folosirea clasei **DesktopPane** este necesară deoarece aceasta ”știe” cum să gestioneze ferestrele interne, având în vedere că acestea se pot suprapune și la un moment dat doar una singură este activă.

Exemplul următor pune în evidență modelul general de creare și afișare a ferestrelor interne:

Listing 11.3: Folosirea ferestrelor interne

```
import javax.swing.*;
import java.awt.*;

class FereastrPrincipala extends JFrame {
    public FereastrPrincipala(String titlu) {
        super(titlu);
        setSize(300, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        FereastrInterna fin1 = new FereastrInterna();
        fin1.setVisible(true);
        FereastrInterna fin2 = new FereastrInterna();
        fin2.setVisible(true);

        JDesktopPane desktop = new JDesktopPane();
        desktop.add(fin1);
        desktop.add(fin2);
        setContentPane(desktop);

        fin2.moveToFront();
    }
}

class FereastrInterna extends JInternalFrame {
    static int n = 0; //nr. de ferestre interne
    static final int x = 30, y = 30;

    public FereastrInterna() {
        super("Document #" + (++n),
```

```
        true, //resizable
        true, //closable
        true, //maximizable
        true); //iconifiable
    setLocation(x*n, y*n);
    setSize(new Dimension(200, 100));
}
}
public class TestInternalFrame {
    public static void main(String args[]) {
        new FereastrPrincipala("Test ferestre interne").show();
    }
}
```

Ferestrele create de acest program vor arăta ca în figura de mai jos:



11.3 Clasa JComponent

`JComponent` este superclasa tuturor componentelor Swing, mai puțin a celor care descriu containere de nivel înalt `JFrame`, `JDialog`, `JApplet`. Deoarece `JComponent` extinde clasa `Container`, deci și `Component`, ea moștenește funcționalitatea generală a containerelor și componentelor AWT, furnizând bineînțeles și o serie întreagă de noi facilități.

Dintre noutățile oferite de `JComponent` amintim:

- **ToolTip**

Folosind metoda `setToolTip` poate fi atașat unei componente un text cu explicații legate de componenta respectivă. Când utilizatorul trece

cu mouse-ul deasupra componentei va fi afișat, pentru o perioadă de timp, textul ajutor specificat.

- **Chenare**

Orice componentă Swing poate avea unul sau mai multe chenare. Specificarea unui chenar se realizează cu metoda `setBorder`.

- **Suport pentru plasare și dimensionare**

Folosind metodele `setPreferredSize`, `setMinimumSize`, `setMaximumSize`, `setAlignmentX`, `setAlignmentY` pot fi controlați parametrii folosiți de gestionarii de poziționare pentru plasarea și dimensionarea automată a componentelor în cadrul unui container.

- **Controlul opacității**

Folosind metoda `setOpaque` vom specifica dacă o componentă trebuie sau nu să deseneze toți pixelii din interiorul său. Implicit, valoarea proprietății de opacitate este `false`, ceea ce înseamnă că este posibil să nu fie desenați unii sau chiar toți pixelii, permițând pixelilor de sub componentă să rămână vizibili (componenta nu este opacă). Valoarea proprietății pentru clasele derivate din `JComponent` depinde în general de Look-and-Feel-ul folosit.

- **Asocierea de acțiuni tastelor**

Pentru componentele Swing există posibilitatea de a specifica anumite acțiuni care să se execute atunci când utilizatorul apasă o anumită combinație de taste și componenta respectivă este activă (are focus-ul). Această facilitate simplifică varianta inițială de lucru, și anume tratarea evenimentelor de tip `KeyEvent` printr-un obiect `KeyListener`.

- **Double-Buffering**

Tehnica de double-buffering, care implică desenarea componentei în memorie și apoi transferul întregului desen pe ecran, este implementată automat de componentele Swing, spre deosebire de cele AWT unde trebuia realizată manual dacă era cazul.

Exemplul următor ilustrează modul de folosire a câtorva dintre facilitățile amintite mai sus:

Listing 11.4: Facilități oferite de clasa `JComponent`

```
import javax.swing.*;
import javax.swing.border.*;
import java.awt.*;
import java.awt.event.*;

class Fereastră extends JFrame {
    public Fereastră(String titlu) {
        super(titlu);
        getContentPane().setLayout(new FlowLayout());
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Folosirea chenarelor
        Border lowered, raised;
        TitledBorder title;

        lowered = BorderFactory.createLoweredBevelBorder();
        raised = BorderFactory.createRaisedBevelBorder();
        title = BorderFactory.createTitledBorder("Borders");

        final JPanel panel = new JPanel();
        panel.setPreferredSize(new Dimension(400,200));
        panel.setBackground(Color.blue);
        panel.setBorder(title);
        getContentPane().add(panel);

        JLabel label1 = new JLabel("Lowered");
        label1.setBorder(lowered);
        panel.add(label1);

        JLabel label2 = new JLabel("Raised");
        label2.setBorder(raised);
        panel.add(label2);

        // Controlul opacitatii
        JButton btn1 = new JButton("Opaque");
        btn1.setOpaque(true); // implicit
        panel.add(btn1);

        JButton btn2 = new JButton("Transparent");
        btn2.setOpaque(false);
        panel.add(btn2);

        // ToolTips
        label1.setToolTipText("Eticheta coborata");
        label2.setToolTipText("Eticheta ridicata");
    }
}
```

```

btn1.setToolTipText("Buton opac");
// Textul poate fi HTML
btn2.setToolTipText("<html><b>Apasati <font color=red>F2
    </font> " +
    "cand butonul are <u>focusul</u>");

// Asocierea unor actiuni (KeyBindings)
/* Apasarea tastei F2 cand focusul este pe butonul al
doilea
va determina schimbarea culorii panelului */
btn2.getInputMap().put(KeyStroke.getKeyStroke("F2"),
    "schimbaCuloare");
btn2.getActionMap().put("schimbaCuloare", new
    AbstractAction() {
    private Color color = Color.red;
    public void actionPerformed(ActionEvent e) {
    panel.setBackground(color);
    color = (color == Color.red ? Color.blue : Color.red)
    ;
    }
});

pack();
}
}

public class TestJComponent {
    public static void main(String args[]) {
    new Fereastra("Facilitati JComponent").show();
    }
}

```

11.4 Arhitectura modelului Swing

11.5 Folosirea modelelor

Modelul Swing este bazat pe o arhitectură asemănătoare cu *MVC* (*model-view-controller*). Arhitectura MVC specifică descompunerea unei aplicații vizuale în trei părți separate:

- *Modelul* - care va reprezenta datele aplicației.

- *Prezentarea* - modul de reprezentare vizuală a datelor.
- *Controlul* - transformarea acțiunilor utilizatorului asupra componentelor vizuale în evenimente care să actualizeze automat modelul acestora (datele).

Din motive practice, în Swing părțile de prezentare și control au fost cuplate deoarece exista o legătură prea strânsă între ele pentru a fi concepute ca entități separate. Așadar, arhitectura Swing este de fapt o arhitectură cu *model separabil*, în care datele componentelor (modelul) sunt separate de reprezentarea lor vizuală. Această abordare este logică și din perspectiva faptului că, în general, modul de concepere a unei aplicații trebuie să fie orientat asupra reprezentării și manipulării informațiilor și nu asupra interfeței grafice cu utilizatorul.

Pentru a realiza separarea modelului de prezentare, fiecărui obiect corespunzător unei clase ce descrie o componentă Swing îi este asociat un obiect care gestionează datele sale și care implementează o interfață care reprezintă modelul componentei respective. După cum se observă din tabelul de mai jos, componente cu reprezentări diferite pot avea același tip de model, dar există și componente care au asociate mai multe modele:

Model	Componentă
ButtonModel	JButton, JToggleButton, JCheckBox, JRadioButton, JMenu, JMenuItem, JCheckBoxMenuItem, JRadioButtomMenuItem
JComboBox	ComboBoxModel
BoundedRangeModel	JProgressBar, JScrollBar, JSlider
JTabbedPane	SingleSelectionModel
ListModel	JList
ListSelectionModel	JList
JTable	TableModel
JTable	TableColumnModel
JTree	TreeModel
JTree	TreeSelectionModel
Document	JEditorPane, JTextPane, JTextArea, JTextField, JPasswordField

Fiecare componentă are un model inițial implicit, însă are posibilitatea de a-l înlocui cu unul nou atunci când este cazul. Metodele care accesează

modelul unui obiect sunt: **setModel**, respectiv **getModel**, cu argumente specifice fiecărei componente în parte. Crearea unei clase care să reprezinte un model se va face extinzând interfața corespunzătoare și implementând metodele definite de aceasta sau extinzând clasa implicită oferită de API-ul Swing și supradefinind metodele care ne interesează. Pentru modelele mai complexe, cum ar fi cele asociate claselor **JTable**, **JTree** sau **JList** există clase abstracte care implementează interfața ce descrie modelul respectiv. De exemplu, interfața model a clasei **JList** este **ListModel** care este implementată de clasele **DefaultListModel** și **AbstractListModel**. În funcție de necesități, oricare din aceste clase poate fi extinsă pentru a crea un nou model.

Listing 11.5: Folosirea mai multor modele pentru o componenta

```
import javax.swing.*;
import javax.swing.border.*;
import java.awt.*;
import java.awt.event.*;

class Fereastră extends JFrame implements ActionListener {
    String data1[] = {"rosu", "galben", "albastru"};
    String data2[] = {"red", "yellow", "blue"};
    int tipModel = 1;
    JList lst;
    ListModel model1, model2;

    public Fereastră(String titlu) {
        super(titlu);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Lista initiala nu are nici un model
        lst = new JList();
        getContentPane().add(lst, BorderLayout.CENTER);

        // La apasara butonului schimbam modelul
        JButton btn = new JButton("Schimba modelul");
        getContentPane().add(btn, BorderLayout.SOUTH);
        btn.addActionListener(this);

        // Cream obiectele corespunzatoare celor doua modele
        model1 = new Model1();
        model2 = new Model2();
        lst.setModel(model1);
    }
}
```

```

    pack();
}

public void actionPerformed(ActionEvent e) {
    if (tipModel == 1) {
        lst.setModel(model2);
        tipModel = 2;
    }
    else {
        lst.setModel(model1);
        tipModel = 1;
    }
}

// Clasele corespunzatoare celor doua modele
class Model1 extends AbstractListModel {
    public int getSize() {
        return data1.length;
    }
    public Object getElementAt(int index) {
        return data1[index];
    }
}

class Model2 extends AbstractListModel {
    public int getSize() {
        return data2.length;
    }
    public Object getElementAt(int index) {
        return data2[index];
    }
}

}

public class TestModel {
    public static void main(String args[]) {
        new Fereastr("Test Model").show();
    }
}

```

Multe componente Swing furnizează metode care să obțină starea obiectului fără a mai fi nevoie să obținem instanța modelului și să apelăm metodele

acesteia. Un exemplu este metoda `getValue` a clasei `JSlider` care este de fapt un apel de genul `getModel().getValue()`. În multe situații însă, mai ales pentru clase cum ar fi `JTable` sau `JTree`, folosirea modelelor aduce flexibilitate sporită programului și este recomandată utilizarea lor.

11.5.1 Tratarea evenimentelor

Modelele componentelor trebuie să notifice apariția unor schimbări ale datelor gestionate astfel încât să poată fi reactualizată prezentarea lor sau să fie executat un anumit cod în cadrul unui obiect de tip *listener*. În Swing, această notificare este realizată în două moduri:

1. Informativ (lightweight) - Modelele trimit un eveniment prin care sunt informați ascultătorii că a survenit o anumită schimbare a datelor, fără a include în eveniment detalii legate de schimbarea survenită. Obiectele de tip listener vor trebui să apeleze metode specifice componentelor pentru a afla *ce anume* s-a schimbat. Acest lucru se realizează prin interfața **ChangeListener** iar evenimentele sunt de tip **ChangeEvent**, modelele care suportă această abordare fiind `BoundedRangeModel`, `ButtonModel` și `SingleSelectionModel`.

Model	Listener	Tip Eveniment
<code>BoundedRangeModel</code>	<code>ChangeListener</code>	<code>ChangeEvent</code>
<code>ButtonModel</code>	<code>ChangeListener</code>	<code>ChangeEvent</code>
<code>SingleSelectionModelModel</code>	<code>ChangeListener</code>	<code>ChangeEvent</code>

Interfața `ChangeListener` are o singură metodă:

```
public void stateChanged(ChangeEvent e),
```

singura informație conținută în eveniment fiind componenta sursă.

Înregistrarea și eliminarea obiectelor de tip listener se realizează cu metodele `addChangeListener`, respectiv `removeChangeListener`.

```
JSlider slider = new JSlider();
BoundedRangeModel model = slider.getModel();
model.addChangeListener(new ChangeListener() {
    public void stateChanged(ChangeEvent e) {
        // Sursa este de tip BoundedRangeModel
        BoundedRangeModel m = (BoundedRangeModel)e.getSource();
        // Trebuie sa interogam sursa asupra schimbarii
```

```

        System.out.println("Schimbare model: " + m.getValue());
    }
});

```

Pentru ușurința programării, pentru a nu lucra direct cu instanța modelului, unele clase permit înregistrarea ascultătorilor direct pentru componenta în sine, singura diferență față de varianta anterioară constând în faptul că sursa evenimentului este acum de tipul componentei și nu de tipul modelului. Secvența de cod de mai sus poate fi rescrisă astfel:

```

JSlider slider = new JSlider();
slider.addChangeListener(new ChangeListener() {
    public void stateChanged(ChangeEvent e) {
        // Sursa este de tip JSlider
        JSlider s = (JSlider)e.getSource();
        System.out.println("Valoare noua: " + s.getValue());
    }
});

```

2. Consistent(statefull) - Modele pun la dispoziție interfețe specializate și tipuri de evenimente specifice ce includ toate informațiile legate de schimbarea datelor.

Model	Listener	Tip Eveniment
ListModel	ListDataListener	ListDataEvent
ListSelectionModel	ListSelectionListener	ListSelectionEvent
ComboBoxModel	ListDataListener	ListDataEvent
TreeModel	TreeModelListener	TreeModelEvent
TreeSelectionModel	TreeSelectionListener	TreeSelectionEvent
TableModel	TableModelListener	TableModelEvent
TableColumnModel	TableColumnModelListener	TableColumnModelEvent
Document	DocumentListener	DocumentEvent
Document	UndoableEditListener	UndoableEditEvent

Folosirea acestor interfețe nu diferă cu nimic de cazul general:

```

String culori[] = {"rosu", "galben", "albastru"};
JList list = new JList(culori);
ListSelectionModel sModel = list.getSelectionModel();
sModel.addListSelectionListener(

```

```
new ListSelectionListener() {
    public void valueChanged(ListSelectionEvent e) {
        // Schimbarea este continuta in eveniment
        if (!e.getValueIsAdjusting()) {
            System.out.println("Selectie curenta: " +
                e.getFirstIndex());
        }
    }
});
```

11.6 Folosirea componentelor

Datorită complexității modelului Swing, în această secțiune nu vom încerca o abordare exhaustivă a modului de utilizare a tuturor componentelor, ci vom pune în evidență doar aspectele specifice acestui model, subliniind diferențele și îmbunătățirile față AWT.

11.6.1 Componente atomice

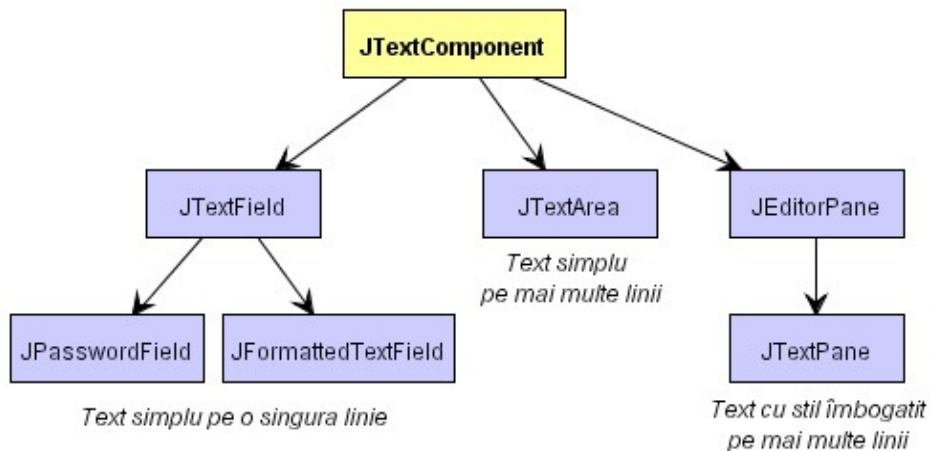
În categoria componentelor atomice includem componentele Swing cu funcționalitate simplă, a căror folosire este în general asemănătoare cu a echivalentelor din AWT. Aici includem:

- Etichete: `JLabel`
- Butoane simple sau cu două stări: `JButton`, `JCheckBox`, `JRadioButton`, `JToggleButton`; mai multe butoane radio pot fi grupate folosind clasa `ButtonGroup`, pentru a permite selectarea doar a unuia dintre ele.
- Componente pentru progres și derulare: `JSlider`, `JProgressBar`, `JScrollBar`
- Separatori: `JSeparator`

Deoarece utilizarea acestora este în general facilă, nu vom analiza în parte aceste componente.

11.6.2 Componente pentru editare de text

Componentele Swing pentru afișarea și editarea textelor sunt grupate într-o ierarhie ce are ca rădăcină clasa `JTextComponent` din pachetul `javax.swing.text`.



După cum se observă din imaginea de mai sus, clasele pot împărțite în trei categorii, corespunzătoare tipului textului editat:

- Text simplu pe o singură linie
 - **JTextField** - Permite editarea unui text simplu, pe o singură linie.
 - **JPasswordField** - Permite editarea de parole. Textul acestora va fi ascuns, în locul caracterelor introduse fiind afișat un caracter simbolic, cum ar fi '*'.
 - **JFormattedTextField** - Permite introducerea unui text care să respecte un anumit format, fiind foarte utilă pentru citirea de numere, date calendaristice, etc. Este folosită împreună cu clase utilitare pentru formatarea textelor, cum ar fi `NumberFormatter`, `DateFormatter`, `MaskFormatter`, etc. Valoarea conținută de o astfel de componentă va fi obținută/setată cu metodele `getValue`, respectiv `setValue` și nu cu cele uzuale `getText`, `setText`.
- Text simplu pe mai multe linii
 - **JTextArea** - Permite editarea unui text simplu, pe mai multe linii. Orice atribut legat de stil, cum ar fi culoarea sau fontul, se aplică întregului text și nu poate fi specificat doar unei anumite porțiuni. Uzual, o componentă de acest tip va fi inclusă într-un container `JScrollPane`, pentru a permite navigarea pe verticală

și orizontală dacă textul introdus nu încapă în suprafața alocată obiectului. Acest lucru este valabil pentru toate componentele Swing pentru care are sens noțiunea de navigare pe orizontală sau verticală, nici una neoferind suport intrinsec pentru această operațiune.

- Text cu stil îmbogățit pe mai multe linii
 - **JEditorPane** - Permite afișarea și editarea de texte scrise cu stiluri multiple și care pot include imagini sau chiar diverse alet componente. Implicit, următoarele tipuri de texte sunt recunoscute: **text/plain**, **text/html** și **text/rtf**. Una din utilizările cele mai simple ale acestei clase este setarea documentului ce va fi afișat cu metoda **setPage**, ce primește ca argument un URL care poate referi un fișier text, HTML sau RTF.
 - **JTextPane** - Această clasă extinde **JEditorPane**, oferind diverse facilități suplimentare pentru lucrul cu stiluri și paragrafe.

Clasa **JTextComponent** încearcă să păstreze cât mai multe similitudini cu clasa **TextComponent** din AWT, însă există diferențe notabile între cele două, componenta Swing având caracteristici mult mai complexe cum ar fi suport pentru operații de *undo* și *redo*, tratarea evenimentelor generate de cursor (caret), etc. Orice obiect derivat din **JTextComponent** este format din:

- Un **model**, referit sub denumirea de *document*, care gestionează starea componentei. O referință la model poate fi obținută cu metoda **getDocument**, ce returnează un obiect de tip **Document**.
- O **reprezentare**, care este responsabilă cu afișarea textului.
- Un **'controller'**, cunoscut sub numele de *editor kit* care permite scrierea și citirea textului și care permite definirea de acțiuni necesare editării.

Există diferenț față de AWT și la nivelul tratării evenimentelor generate de componentele pentru editarea de texte. Dintre evenimentele ce pot fi generate amintim:

- **ActionEvent** - Componentele derivate din **JTextField** vor genera un eveniment de acest tip la apăsarea tastei Enter în căsuța de editare a textului. Interfața care trebuie implementată este **ActionListener**.

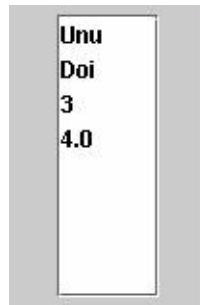
- **CaretEvent** - Este evenimentul generat la deplasarea cursorului ce gestionează poziția curentă în text. Interfața corespunzătoare **CaretListener** conține o singură metodă: **caretUpdate** ce va fi apelată ori de câte ori apare o schimbare.
- **DocumentEvent** - Evenimentele de acest tip sunt generate la orice schimbare a textului, sursa lor fiind documentul (modelul) componente și nu componenta în sine. Interfața corespunzătoare este **DocumentListener**, ce conține metodele:
 - **insertUpdate** - apelată la adăugarea de noi caractere;
 - **removeUpdate** - apelată după o operațiune de ștergere;
 - **changedUpdate** - apelată la schimbarea unor atribute legate de stilul textului.
- **PropertyChangeEvent** - Este un eveniment comun tuturor componentelor de tip **JavaBean**, fiind generat la orice schimbare a unei proprietăți a componente. Interfața corespunzătoare este **PropertyChangeListener**, ce conține metoda **propertyChange**.

11.6.3 Componente pentru selectarea unor elemente

În această categorie vom include clasele care permit selectarea unor valori (elemente) dintr-o serie prestabilită. Acestea sunt: **JList**, **JComboBox** și **JSpinner**.

Clasa **JList**

Clasa **JList** descrie o listă de elemente dispuse pe una sau mai multe coloane, din care utilizatorul poate selecta unul sau mai multe. Uzual un obiect de acest tip va fi inclus într-un container de tip **JScrollPane**.



Inițializarea unei liste se realizează în mai multe modalități:

- Folosind unul din constructorii care primesc ca argument un vector de elemente.

```
Object elemente[] = {"Unu", "Doi", new Integer(3), new Double(4)};
JList lista = new JList(elemente);
```

- Folosind constructorul fără argumente și adăugând apoi elemente modelului implicit listei:

```
DefaultListModel model = new DefaultListModel();
model.addElement("Unu");
model.addElement("Doi");
model.addElement(new Integer(3));
model.addElement(new Double(4));
JList lista = new JList(model);
```

- Folosind un model propriu, responsabil cu furnizarea elementelor listei. Acesta este un obiect dintr-o clasă ce trebuie să implementeze interfața **ListModel**, uzual fiind folosită extinderea clasei predefinite **AbstractListModel** și supradefinirea metodelor: **getElementAt** care furnizează elementul de pe o anumită poziție din listă, respectiv **getSize** care trebuie să returneze numărul total de elemente din listă. Evident, această variantă este mai complexă, oferind flexibilitate sporită în lucrul cu liste.

```
ModelLista model = new ModelLista();
JList lista = new JList(model);
```

```

...
class ModelLista extends AbstractListModel {
    Object elemente[] = {"Unu", "Doi", new Integer(3), new Double(4)};
    public int getSize() {
        return elemente.length;
    }
    public Object getElementAt(int index) {
        return elemente[index];
    }
}

```

Gestiunea articolelor selectate dintr-o listă se realizează prin intermediul unui model, acesta fiind un obiect de tip **ListSelectionModel**. Obiectele de tip **JList** generează evenimente de tip **ListSelectionEvent**, interfața corespunzătoare fiind **ListSelectionListener** ce conține metoda **valueChanged** apelată ori de câte ori va fi schimbată selecția elementelor din listă.

```

class Test implements ListSelectionListener {
    ...
    public Test() {
        ...
        // Stabilim modul de selectie
        list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        /* sau SINGLE_INTERVAL_SELECTION
           MULTIPLE_INTERVAL_SELECTION
        */

        // Adaugam un ascultator
        ListSelectionModel model = list.getSelectionModel();
        model.addListSelectionListener(this);
        ...
    }

    public void valueChanged(ListSelectionEvent e) {
        if (e.getValueIsAdjusting()) return;
        int index = list.getSelectedIndex();
    }
}

```

```

    ...
  }
}

```

Evident, clasa oferă metode pentru selectarea unor elemente din cadrul programului `setSelectedIndex`, `setSelectedIndices`, etc. și pentru obținerea celor selectate la un moment dat `getSelectedIndex`, `getSelectedIndices`, etc..

O facilitate extrem de importantă pe care o au listele este posibilitatea de a stabili un *renderer* pentru fiecare articol în parte. Implicit toate elementele listei sunt afișate în același fel, însă acest lucru poate fi schimbat prin crearea unei clase ce implementează interfața **ListCellRenderer** și personalizează reprezentarea elementelor listei în funcție de diverși parametri. Interfața `ListCellRenderer` conține o singură metodă `getListCellRendererComponent` ce returnează un obiect de tip `Component`. Metoda va fi apelată în parte pentru reprezentarea fiecărui element al listei.

```

class MyCellRenderer extends JLabel implements ListCellRenderer {
    public MyCellRenderer() {
        setOpaque(true);
    }
    public Component getListCellRendererComponent(
        JList list, Object value, int index,
        boolean isSelected, boolean cellHasFocus) {
        setText(value.toString());
        setBackground(isSelected ? Color.red : Color.white);
        setForeground(isSelected ? Color.white : Color.black);
        return this;
    }
}

```

Setarea unui anumit *renderer* pentru o listă se realizează cu metoda `setCellRenderer`.

Clasa `JComboBox`

Clasa `JComboBox` este similară cu `JList`, cu deosebirea că permite doar selectarea unui singur articol, acesta fiind și singurul permanent vizibil. Lista

celorlalte elemente este afișată doar la apăsarea unui buton marcat cu o săgeată, ce face parte integrantă din componentă.



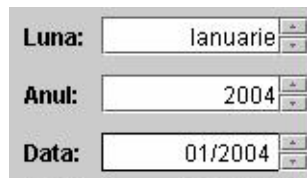
JComboBox funcționează după aceleași principii ca și clasa JList. Inițializarea se face dintr-un vector sau folosind un model de tipul **ComboBoxModel**, fiecare element putând fi de asemenea reprezentat diferit prin intermediul unui obiect ce implementează aceeași interfață ca și în cazul listelor: **ListCellRenderer**.

O diferență notabilă constă în modul de selectare a unui articol, deoarece JComboBox permite și editarea explicită a valorii elementului, acest lucru fiind controlat de metoda **setEditable**.

Evenimentele generate de obiectele JComboBox sunt de tip **ItemEvent** generate la navigarea prin listă, respectiv **ActionEvent** generate la selectarea efectivă a unui articol.

Clasa JSpinner

Clasa JSpinner oferă posibilitatea de a selecta o anumită valoare (element) dintr-un domeniu prestabilit, lista elementelor nefiind însă vizibilă. Este folosit atunci când domeniul din care poate fi făcută selecția este foarte mare sau chiar nemărginit; de exemplu: numere întregi între 1950 și 2050. Componenta conține două butoane cu care poate fi selectat următorul, respectiv predecesorul element din domeniu.



JSpinner se bazează exclusiv pe folosirea unui model. Acesta este un obiect de tip **SpinnerModel**, existând o serie de clase predefinite ce implementează această interfață cum ar fi **SpinnerListModel**, **SpinnerNumberModel** sau **SpinnerDateModel** ce pot fi utilizate.

Componentele de acest tip permit și specificarea unui anumit tip de editor pentru valorile elementelor sale. Acesta este instalat automat pentru fiecare din modelele standard amintite mai sus, fiind reprezentat de una din clasele `JSpinner.ListEditor`, `JSpinner.NumberEditor`, respectiv `JSpinner.DateEditor`, toate derivate din `JSpinner.DefaultEditor`. Fiecare din editoarele amintite permite diverse formătări specifice.

Evenimentele generate de obiectele de tip `JSpinner` sunt de tip `ChangeEvent`, generate la schimbarea stării componentei.

11.6.4 Tabele

Clasa `JTable` permite crearea de componente care să afișeze o serie de elemente într-un format tabelar, articolele fiind dispuse pe linii și coloane. Un tabel poate fi folosit doar pentru afișarea formatată a unor date, dar este posibilă și editarea informației din celulele sale. De asemenea, liniile tabelului pot fi marcate ca selectate, tipul selecției fiind simplu sau compus, tabelele extinzând astfel funcționalitatea listelor.

Nume	Varsta	Student
Ionescu	20	true
Popescu	80	false

Deși clasa `JTable` se găsește în pachetul `javax.swing`, o serie de clase și interfețe necesare lucrului cu tabele se găsesc în pachetul `javax.swing.table`, acesta trebuind așadar importat.

Inițializarea unui tabel poate fi făcută în mai multe moduri. Cea mai simplă variantă este să folosim unul din constructorii care primesc ca argumente elementele tabelului sub forma unei matrici sau a unei colecții de tip `Vector` și denumirile capurilor de coloană:

```
String[] coloane = {"Nume", "Varsta", "Student"};
Object[][] elemente = {
    {"Ionescu", new Integer(20), Boolean.TRUE},
    {"Popescu", new Integer(80), Boolean.FALSE}};
JTable tabel = new JTable(elemente, coloane);
```

După cum se observă, tipul de date al elementelor de pe o coloană este de tip referință și poate fi oricare. În cazul în care celulele tabelului sunt

editabile trebuie să existe un editor potrivit pentru tipul elementului din celula respectivă. Din motive de eficiență, implementarea acestei clase este orientată la nivel de coloană, ceea ce înseamnă că articole de pe o coloană vor fi reprezentate la fel și vor avea același tip de editor.

A doua variantă de creare a unui tabel este prin implementarea modelului acestuia într-o clasă separată și folosirea constructorului corespunzător. Interfața care descrie modelul clasei `JTable` este `TableModel` și conține metodele care vor fi interogate pentru obținerea informației din tabel. Uzual, crearea unui model se face prin extinderea clasei predefinite `AbstractTableModel`, care implementează deja `TableModel`. Tot ceea ce trebuie să facem este să supradefinim metodele care ne interesează, cele mai utilizate fiind (primele trei trebuie obligatoriu supradefinite, ele fiind declarate abstracte în clasa de bază):

- `getRowCount` - returnează numărul de linii ale tabelului;
- `getColumnCount` - returnează numărul de coloane ale tabelului;
- `getValueAt` - returnează elementul de la o anumită linie și coloană;
- `columnName` - returnează denumirea fiecărei coloane;
- `isCellEditable` - specifică dacă o anumită celulă este editabilă.

Modelul mai conține și metoda `setValueAt` care poate fi folosită pentru setarea explicită a valorii unei celule.

```

ModelTabel model = new ModelTabel();
JTable tabel = new JTable(model);
...
class ModelTabel extends AbstractTableModel {
    String[] coloane = {"Nume", "Varsta", "Student"};
    Object[][] elemente = {
        {"Ionescu", new Integer(20), Boolean.TRUE},
        {"Popescu", new Integer(80), Boolean.FALSE}};

    public int getColumnCount() {
        return coloane.length;
    }
    public int getRowCount() {

```

```

        return elemente.length;
    }
    public Object getValueAt(int row, int col) {
        return elemente[row][col];
    }
    public String getColumnName(int col) {
        return coloane[col];
    }
    public boolean isCellEditable(int row, int col) {
        // Doar numele este editabil
        return (col == 0);
    }
}

```

Orice schimbare a datelor tabelului va genera un eveniment de tip **TableModelEvent**. Pentru a trata aceste evenimente va trebui să implementăm interfața **TableModelListener** ce conține metoda **tableChanged**. Inregistrarea unui listener va fi făcută pentru modelul tabelului:

```

public class Test implements TableModelListener {
    ...
    public Test() {
        ...
        tabel.getModel().addTableModelListener(this);
        ...
    }
    public void tableChanged(TableModelEvent e) {
        // Aflam celula care a fost modificata
        int row = e.getFirstRow();
        int col = e.getColumn();
        TableModel model = (TableModel)e.getSource();
        Object data = model.getValueAt(row, col);
        ...
    }
}

```

Tabele oferă posibilitatea de a selecta una sau mai multe linii, nu neapărat consecutive, gestiunea liniilor selectate fiind realizată prin intermediul unui model. Acesta este o instanță ce implementează, întocmai ca la liste, interfața **ListSelectionModel**. Tratarea evenimentelor generate de schimbarea selecției în tabel se realizează prin înregistrarea unui ascultător de tip **ListSelectionListener**:

```
class Test implements ListSelectionListener {
    ...
    public Test() {
        ...
        // Stabilim modul de selectie
        tabel.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

        // Adaugam un ascultator
        ListSelectionModel model = tabel.getSelectionModel();
        model.addListSelectionListener(this);
        ...
    }

    public void valueChanged(ListSelectionEvent e) {
        if (e.getValueIsAdjusting()) return;
        ListSelectionModel model =
            (ListSelectionModel)e.getSource();
        if (model.isEmpty()) {
            // Nu este nici o linie selectata
            ...
        } else {
            int index = model.getMinSelectionIndex();
            // Linia cu numarul index este prima selectata
            ...
        }
    }
}
```

După cum am spus, celulele unei coloane vor fi reprezentate la fel, fiecare coloană având asociat un obiect *renderer* responsabil cu crearea componen-

tei ce descrie celulele sale. Un astfel de obiect implementează interfața **TableCellRenderer**, care are o singură metodă **getTableCellRendererComponent**, aceasta fiind responsabilă cu crearea componentelor ce vor fi afișate în celulele unei coloane. Implicit, există o serie de tipuri de date cu reprezentări specifice, cum ar fi: **Boolean**, **Number**, **Double**, **Float**, **Date**, **ImageIcon**, **Icon**, restul tipurilor având o reprezentare standard ce constă într-o etichetă cu reprezentarea obiectului ca șir de caractere. Specificarea unui *renderer* propriu se realizează cu metoda **setDefaultRenderer**, ce asociază un anumit tip de date cu un obiect de tip **TableRenderer**.

```
public class MyRenderer extends JLabel
    implements TableCellRenderer {
    public Component getTableCellRendererComponent(...) {
        ...
        return this;
    }
}
```

O situație similară o regăsim la nivelul editorului asociat celulelor dintr-o anumită coloană. Acesta este un obiect ce implementează interfața **TableCellEditor**, ce extinde interfața **CellEditor** care generalizează conceptul de celulă editabilă pe care îl vom mai regăsi la arbori. Implicit, există o serie de editoare standard pentru tipurile de date menționate anterior, dar este posibilă specificarea unui editor propriu cu metoda **setDefaultEditor**. Crearea unui editor propriu se realizează cel mai simplu prin extinderea clasei utilitare **AbstractCellEditor**, care implementează **CellEditor**, plus implementarea metodei specifice din **TableCellEditor**.

```
public class MyEditor
    extends AbstractCellEditor
    implements TableCellEditor {

    // Singura metoda abstracta a parintelui
    public Object getCellEditorValue() {
        // Returneaza valoarea editata
        ...
    }
}
```

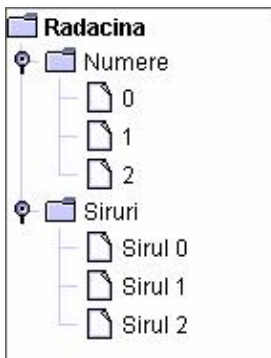
```

// Metoda definita de TableCellEditor
public Component getTableCellEditorComponent(...) {
    // Returneaza componenta de tip editor
    ...
}
}

```

11.6.5 Arbori

Clasa **JTree** permite afișarea unor elemente într-o manieră ierarhică. Ca orice componentă Swing netrivială, un obiect **JTree** reprezintă doar o imagine a datelor, informația în sine fiind manipulată prin intermediul unui model. La nivel structural, un arbore este format dintr-o *rădăcină*, *noduri interne* - care au cel puțin un fiu și *noduri frunză* - care nu mai au nici un descendent.



Deși clasa **JTree** se găsește în pachetul `javax.swing`, o serie de clase și interfețe necesare lucrului cu arbori se găsesc în pachetul `javax.swing.tree`.

Clasa care modelează noțiunea de nod al arborelui este **DefaultMutableTreeNode**, aceasta fiind folosită pentru toate tipurile de noduri. Crearea unui arbore presupune așadar crearea unui nod (rădăcina), instanțierea unui obiect de tip **JTree** cu rădăcina creată și adăugarea apoi de noduri frunză ca fii ai unor noduri existente.

```

String text = "<html><b>Radacina</b></html>";
DefaultMutableTreeNode root = new DefaultMutableTreeNode(text);
DefaultMutableTreeNode numere =
    new DefaultMutableTreeNode("Numere");
DefaultMutableTreeNode siruri =

```

```

        new DefaultMutableTreeNode("Siruri");

    for(int i=0; i<3; i++) {
        numere.add(new DefaultMutableTreeNode(new Integer(i)));
        siruri.add(new DefaultMutableTreeNode("Sirul " + i));
    }

    root.add(numere);
    root.add(siruri);

    JTree tree = new JTree(root);

```

După cum se observă, nodurile arborelui pot fi de tipuri diferite, reprezentarea lor implicită fiind obținută prin apelarea metodei `toString` pentru obiectului conținut. De asemenea, este posibilă specificarea unui text în format HTML ca valoare a unui nod, acesta fiind reprezentat ca atare.

Dacă varianta adăugării explicite a nodurilor nu este potrivită, se poate implementa o clasă care să descrie modelul arborelui. Aceasta trebuie să implementeze interfața **TreeModel**.

Scopul unei componente de tip arbore este în general selectarea unui nod al ierarhiei. Ca și în cazul listelor sau a tabelor, gestiunea elementelor selectate se realizează printr-un model, în această situație interfața corespunzătoare fiind **TreeSelectionModel**. Arborii permit înregistrarea unor obiecte *listener*, de tip **TreeSelectionListener**, care să trateze evenimentele generate la schimbarea selecției în arbore.

```

class Test implements TreeSelectionListener {
    ...
    public Test() {
        ...
        // Stabilim modul de selectie
        tree.getSelectionModel().setSelectionMode(
            TreeSelectionModel.SINGLE_TREE_SELECTION);

        // Adaugam un ascultator
        tree.addTreeSelectionListener(this);
        ...
    }
}

```

```

public void valueChanged(TreeSelectionEvent e) {
    // Obținem nodul selectat
    DefaultMutableTreeNode node = (DefaultMutableTreeNode)
        tree.getLastSelectedPathComponent();

    if (node == null) return;

    // Obținem informația din nod
    Object nodeInfo = node.getUserObject();
    ...
}
}

```

Fiecare nod al arborelui este reprezentat prin intermediul unei clase *renderer*. Aceasta implementează interfața **TreeCellRenderer**, cea folosită implicit fiind **DefaultTreeCellRenderer**. Prin implementarea interfeței sau extinderea clasei implicite pot fi create modalități de personalizare a nodurilor arborelui în funcție de tipul sau valoarea acestora.

Există însă și diverse metode de a schimba înfățișarea unui arbore fără să creăm noi clase de tip **TreeCellRenderer**. Acestea sunt:

- **setRootVisible** - Specifică dacă rădăcina e vizibilă sau nu;
- **setShowsRootHandles** - Specifică dacă nodurile de pe primul nivel au simboluri care să permită expandarea sau restrângerea lor.
- **putClientProperty** - Stabilește diverse proprietăți, cum ar fi modul de reprezentare a relațiilor (liniilor) dintre nodurile părinte și fiu:

```

tree.putClientProperty("JTree.lineStyle", "Angled");
// sau "Horizontal", "None"

```

- Specificarea unei iconițe pentru nodurile frunză sau interne:

```

ImageIcon leaf = createImageIcon("img/leaf.gif");
ImageIcon open = createImageIcon("img/open.gif");
ImageIcon closed = createImageIcon("img/closed.gif");

```

```
DefaultTreeCellRenderer renderer =
    new DefaultTreeCellRenderer();
renderer.setLeafIcon(leaf);
renderer.setOpenIcon(open);
renderer.setClosedIcon(closed);

tree.setCellRenderer(renderer);
```

11.6.6 Containere

După cum știm, containerele reprezintă suprafețe de afișare pe care pot fi plasate ale componente, eventual chiar alte containere. Superclasa componentelor de acest tip este **Container**, clasă despre care am mai discutat în capitolul dedicat modelului AWT.

Containerele pot fi împărțite în două categorii:

1. **Containere de nivel înalt** - Acestea sunt `JFrame`, `JDialog`, `JApplet` și reprezintă rădăcinile ierarhiilor de componente ale unei aplicații.
2. **Containere intermediare** - Reprezintă suprafețe de afișare cu ajutorul cărora pot fi organizate mai eficient componentele aplicației, putând fi imbricate. Cele mai importante clase care descriu astfel de containere sunt:

- `JPanel`
- `JScrollPane`
- `JTabbedPane`
- `JSplitPane`
- `JLayeredPane`
- `JDesktopPane`
- `JRootPane`

JPanel are aceeași funcționalitate ca și clasa `Panel` din AWT, fiind folosit pentru gruparea mai multor componente Swing și plasarea lor împreună pe o altă suprafață de afișare. Gestionarul de poziționare implicit este `FlowLayout`, acesta putând fi schimbat însă chiar în momentul construirii

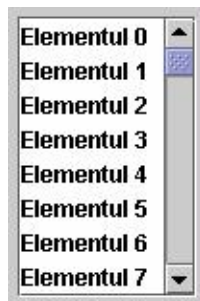
obiectului `JPanel` sau ulterior cu metoda `setLayout`. Adăugarea de componente se realizează ca pentru orice container, folosind metoda `add`.

```
JPanel p = new JPanel(new BorderLayout());
/* Preferabil, deoarece nu mai este construit si
   un obiect de tip FlowLayout (implicit)
*/
p.add(new JLabel("Hello"));
p.add(new JButton("OK"));
...
```

JScrollPane este o clasă foarte importantă în arhitectura modelului Swing, deoarece oferă suport pentru derularea pe orizontală și verticală a componentelor a căror reprezentare completă nu încapă în suprafața asociată, nici o componentă Swing neoferind suport intrinsec pentru această operație.

```
String elemente[] = new String[100];
for(int i=0; i<100; i++)
    elemente[i] = "Elementul " + i;

JList lista = new JList(elemente);
JScrollPane sp = new JScrollPane(lista);
frame.getContentPane().add(sp);
```



JTabbedPane este utilă pentru suprapunerea mai multor containere, uzual panouri (obiecte de tip `JPanel`), pe același spațiu de afișare, selectarea

unuia sau altui panou realizându-se prin intermediul unor butoane dispuse pe partea superioară a componentei, fiecare panou având un astfel de buton corespunzător. Ca funcționalitate, oferă o implementare asemănătoare gestionarului de poziționare `CardLayout`.

```
JTabbedPane tabbedPane = new JTabbedPane();
ImageIcon icon = new ImageIcon("smiley.gif");

JComponent panel1 = new JPanel();
panel1.setOpaque(true);
panel1.add(new JLabel("Hello"));
tabbedPane.addTab("Tab 1", icon, panel1,
    "Aici avem o eticheta");
tabbedPane.setMnemonicAt(0, KeyEvent.VK_1);

JComponent panel2 = new JPanel();
panel2.setOpaque(true);
panel2.add(new JButton("OK"));
tabbedPane.addTab("Tab 2", icon, panel2,
    "Aici avem un buton");
tabbedPane.setMnemonicAt(1, KeyEvent.VK_2);
```



JSplitPane permite crearea unui container care conține două componente dispuse fie una lângă cealaltă, fie una sub alta și separarea acestora prin intermediul unei bare care să permită configurarea suprafeței alocate fiecărei componente.

```
String elem[] = {"Unu", "Doi", "Trei" };
JList list = new JList(elem);
```

```

JPanel panel = new JPanel(new GridLayout(3, 1));
panel.add(new JButton("Adauga"));
panel.add(new JButton("Sterge"));
panel.add(new JButton("Salveaza"));

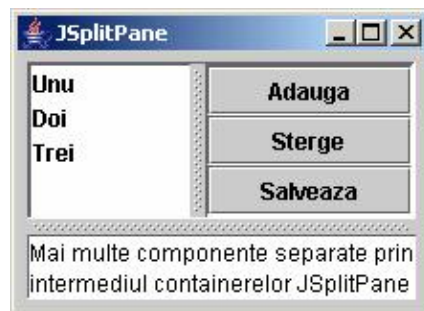
JTextArea text = new JTextArea(
    "Mai multe componente separate prin\n" +
    "intermediul containerelor JSplitPane");

// Separam lista de grupul celor trei butoane
JSplitPane sp1 = new JSplitPane(
    JSplitPane.HORIZONTAL_SPLIT, list, panel);

// Separam containerul cu lista si butoanele
// de componenta pentru editare de text
JSplitPane sp2 = new JSplitPane(
    JSplitPane.VERTICAL_SPLIT, sp1, text);

frame.getContentPane().add(sp2);

```



11.6.7 Dialoguri

Clasa care descrie ferestre de dialog este `JDialog`, crearea unui dialog realizându-se prin extinderea acesteia, întocmai ca în modelul AWT. În Swing există însă o serie de clase predefinite ce descriu anumite tipuri de dialoguri, extrem de utile în majoritatea aplicațiilor. Acestea sunt:

- **JOptionPane** - Permite crearea unor dialoguri simple, folosite pentru afișarea unor mesaje, realizarea unor interogări de confirmare/renunțare,

etc. sau chiar pentru introducerea unor valori, clasa fiind extrem de configurabilă. Mai jos, sunt exemplificate două modalități de utilizare a clasei:

```
JOptionPane.showMessageDialog(frame,  
    "Eroare de sistem !", "Eroare",  
    JOptionPane.ERROR_MESSAGE);  
  
JOptionPane.showConfirmDialog(frame,  
    "Doriti inchiderea aplicatiei ? ", "Intrebare",  
    JOptionPane.YES_NO_OPTION,  
    JOptionPane.QUESTION_MESSAGE);
```

- **JFileChooser** - Dialog standard care permite navigarea prin sistemul de fișiere și selectarea unui anumit fișier pentru operații de deschidere, respectiv salvare.
- **JColorChooser** - Dialog standard pentru selectarea într-o manieră facilă a unei culori.
- **ProgressMonitor** - Clasă utilizată pentru monitorizarea progresului unei operații consumatoare de timp.

11.7 Desenarea

11.7.1 Metode specifice

După cum știm, desenarea unei componente este un proces care se executa automat ori de câte ori este necesar. Procesul în sine este asemănător celui din modelul AWT, însă există unele diferențe care trebuie menționate.

Orice componentă se găsește într-o ierarhie formată de containere, rădăcina acestei fiind un container de nivel înalt, cum ar fi o fereastră sau suprafața unui applet. Cu alte cuvinte, componenta este plasată pe o suprafață de afișare, care la rândul ei poate fi plasată pe altă suprafață și așa mai departe. Când este necesară desenarea componentei respective, fie la prima sa afișare, fie ca urmare a unor acțiuni externe sau interne programului, operația de desenare va fi executată pentru toate containerele, începând cu cel de la nivelul superior.

Desenarea se bazează pe modelul AWT, metoda cea mai importantă fiind **paint**, apelată automat ori de câte ori este necesar. Pentru componentele Swing, această metodă are însă o implementare specifică **și nu trebuie supradefinită**. Aceasta este responsabilă cu apelul metodelor Swing ce desenează componenta și anume:

- **paintComponent** - Este principala metodă pentru desenare ce este supradefinită pentru fiecare componentă Swing în parte pentru a descrie reprezentarea sa grafică. Implicit, în cazul în care componenta este opacă metoda desenează suprafața sa cu culoarea de fundal, după care va executa desenarea propriu-zisă.
- **paintBorder** - Desenează chenarele componentei (dacă există). Nu trebuie supradefinită.
- **paintChildren** - Solicită desenarea componentelor conținute de această componentă (dacă există). Nu trebuie supradefinită.

Metoda **paint** este responsabilă cu apelul metodelor amintite mai sus și realizarea unor optimizări legate de procesul de desenare, cum ar fi implementarea mecanismului de *double-buffering*. Deși este posibilă supradefinirea ei, acest lucru nu este recomandat, din motivele amintite mai sus.

Ca și în AWT, dacă se dorește redesenarea explicită a unei componente se va apela metoda **repaint**. În cazul în care dimensiunea sau poziția componentei s-au schimbat, apelul metodei **revalidate** va precede apelul lui **repaint**.

Atenție

Intocmai ca în AWT, desenarea este realizată de firul de execuție care se ocupă cu transmiterea evenimentelor. Pe perioada în care acesta este ocupat cu transmiterea unui mesaj nu va fi făcută nici o desenare. De asemenea, dacă acesta este blocat într-o operațiune de desenare ce consumă mult timp, pe perioada respectivă nu va fi transmis nici un mesaj.

11.7.2 Considerații generale

În continuare vom prezenta câteva considerații generale legate de diferite aspecte ale desenării în cadrul modelului Swing.

Afișarea imaginilor

În AWT afișarea unei imagini era realizată uzual prin supradefinirea clasei `Canvas` și desenarea imaginii în metoda `paint` a acesteia. În Swing, există câteva soluții mai simple pentru afișarea unei imagini, cea mai utilizată fiind crearea unei etichete (`JLabel`) sau a unui buton (`JButton`) care să aibă setată o anumită imagine pe suprafața sa. Imaginea respectivă trebuie creată folosind clasa `ImageIcon`.

```
ImageIcon img = new ImageIcon("smiley.gif");
JLabel label = new JLabel(img);
```

Transparența

Cu ajutorul metodei `setOpaque` poate fi controlată opacitatea componentelor Swing. Aceasta este o facilitate extrem de importantă deoarece permite crearea de componente care nu au formă rectangulară. De exemplu, un buton circular va fi construit ca fiind transparent (`setOpaque(false)`) și va desena în interiorul său o elipsă umplută cu o anumită culoare. Evident, este necesară implementarea de cod specific pentru a trata apăsarea acestui tip de buton.

Transparența însă vine cu un anumit preț, deoarece pentru componentele transparente vor trebui redesenate containerele pe care se găsește aceasta, încetinind astfel procesul de afișare. Din acest motiv, de fiecare dată când este cazul, se recomandă setarea componentelor ca fiind opace (`setOpaque(true)`).

Dimensiunile componentelor

După cum știm, orice componentă este definită de o suprafață rectangulară. Dimensiunile acestei pot fi obținute cu metodele `getSize`, `getWidth`, `getHeight`. Acestea includ însă și dimensiunile chenarelor, evident dacă acestea există. Suprafața ocupată de acestea poate fi aflată cu metoda `getInsets`

ce va returna un obiect de tip **Insets** ce specifică numărul de pixeli ocupați cu chenare în jurul componentei.

```
public void paintComponent(Graphics g) {
    ...
    Insets insets = getInsets();
    int currentWidth = getWidth() - insets.left - insets.right;
    int currentHeight = getHeight() - insets.top - insets.bottom;
    ...
}
```

Contexte grafice

Argumentul metodei `paintComponent` este de tip **Graphics** ce oferă primitivele standard de desenare. În majoritatea cazurilor însă, argumentul este de fapt de tip **Graphics2D**, clasă ce extinde **Graphics** și pune la dispoziție metode mai sofisticate de desenare cunoscute sub numele de Java2D. Pentru a avea acces la API-ul Java2D, este suficient să facem conversia argumentului ce descrie contextul grafic:

```
public void paintComponent(Graphics g) {
    Graphics2D g2d = (Graphics2D)g;
    // Desenam apoi cu g2d
    ...
}
```

În Swing, pentru a eficientiza desenarea, obiectul de tip **Graphics** primit ca argument de metoda `paintComponent` este refolosit pentru desenarea componentei, a chenarelor și a fiilor săi. Din acest motiv este foarte important ca atunci când supradefinim metoda `paintComponent` să ne asigurăm că la terminarea metodei starea obiectului **Graphics** este aceeași ca la început. Acest lucru poate fi realizat fie explicit, fie folosind o copie a contextului grafic primit ca argument:

```
// 1.Explicit
Graphics2D g2d = (Graphics2D)g;
g2d.translate(x, y); // modificam contextul
...
g2d.translate(-x, -y); // revenim la starea initiala
```

```
// 2. Folosirea unei copii
Graphics2D g2d = (Graphics2D)g.create();
g2d.translate(x, y);
...
g2d.dispose();
```

11.8 Look and Feel

Prin sintagma *'Look and Feel'* (*L&F*) vom înțelege modul în care sunt desenate componentele Swing și felul în care acestea interacționează cu utilizatorul. Posibilitatea de a alege între diferite moduri L&F are avantajul de a oferi prezentarea unei aplicații într-o formă grafică care să corespundă preferințelor utilizatorilor. În principiu, variantele originale de L&F furnizate în distribuția standard ofereau modalitatea ca o interfață Swing fie să se încadreze în ansamblul grafic al sistemului de operare folosit, fie să aibă un aspect specific Java.

Orice L&F este descris de o clasă derivată din **LookAndFeel**. Distribuția standard Java include următoarele clase ce pot fi utilizate pentru selectarea unui L&F:

- `javax.swing.plaf.metal.MetalLookAndFeel`
Este varianta implicită de L&F și are un aspect specific Java.
- `com.sun.java.swing.plaf.windows.WindowsLookAndFeel`
Varianta specifică sistemelor de operare Windows. Incepând cu versiunea 1.4.2 există și implementarea pentru Windows XP .
- `com.sun.java.swing.plaf.mac.MacLookAndFeel`
Varianta specifică sistemelor de operare Mac.
- `com.sun.java.swing.plaf.motif.MotifLookAndFeel`
Specifică interfața CDE/Motif.
- `com.sun.java.swing.plaf.gtk.GTKLookAndFeel`
GTK+ reprezintă un standard de creare a interfețelor grafice dezvoltat independent de limbajul Java. (GTK este acronimul de la *GNU Image Manipulation Program Toolkit*). Folosind acest L&F este posibilă și

specificarea unei anumite teme prin intermediul unui fișier de resurse sau folosind variabila `swing.gtkthemefile`, ca în exemplul de mai jos:

```
java -Dswing.gtkthemefile=temaSpecifica/gtkrc App
```

Specificare unei anumite interfețe L&F poate fi realizată prin mai multe modalități.

Folosirea clasei `UIManager`

Clasa `UIManager` pune la dispoziție o serie de metode statice pentru selectarea la momentul execuției a unui anumit L&F, precum și pentru obținerea unor variante specifice:

- `getLookAndFeel` - Obține varianta curentă, returnând un obiect de tip `LookAndFeel`.
- `setLookAndFeel` - Setează modul curent L&F. Metoda primește ca argument un obiect dintr-o clasă derivată din `LookAndFeel`, fie un șir de caractere cu numele complet al clasei L&F.
- `getSystemLookAndFeelClassName` - Obține varianta specifică sistemului de operare folosit. În cazul în care nu există nici o astfel de clasă, returnează varianta standard.
- `getCrossPlatformLookAndFeelClassName` - Returnează interfața grafică standard Java (JLF).

```
// Exemple:  
UIManager.setLookAndFeel(  
    "com.sun.java.swing.plaf.motif.MotifLookAndFeel");  
  
UIManager.setLookAndFeel(  
    UIManager.getSystemLookAndFeelClassName());
```

Setarea proprietății `swing.defaultlaf`

Există posibilitatea de a specifica varianta de L&F a aplicație direct de la linia de comandă prin setarea proprietății `swing.defaultlaf`:

```
java -Dswing.defaultlaf=  
    com.sun.java.swing.plaf.gtk.GTKLookAndFeel App  
java -Dswing.defaultlaf=  
    com.sun.java.swing.plaf.windows.WindowsLookAndFeel App
```

O altă variantă de a seta această proprietate este schimbarea ei direct în fișierul `swing.properties` situat în subdirectorul `lib` al distribuției Java.

```
# Swing properties  
swing.defaultlaf=  
    com.sun.java.swing.plaf.windows.WindowsLookAndFeel
```

Ordinea în care este aleasă clasa L&F este următoarea:

1. Apelul explicit al metodei `UIManager.setLookAndFeel` înaintea creării unei componente Swing.
2. Proprietatea `swing.defaultlaf` specificată de la linia de comandă.
3. Proprietatea `swing.defaultlaf` specificată în fișierul `swing.properties`.
4. Clasa standard Java (JLF).

Există posibilitatea de a schimba varianta de L&F chiar și după afișarea componentelor. Acesta este un proces care trebuie să actualizeze ierarhiile de componente, începând cu containerele de nivel înalt și va fi realizat prin apelul metodei `SwingUtilities.updateComponentTreeUI` ce va primi ca argument rădăcina unei ierarhii. Secvența care efectuează această operație pentru o fereastră *f* este:

```
UIManager.setLookAndFeel( numeClasaLF );  
SwingUtilities.updateComponentTreeUI(f);  
f.pack();
```

Capitolul 12

Fire de execuție

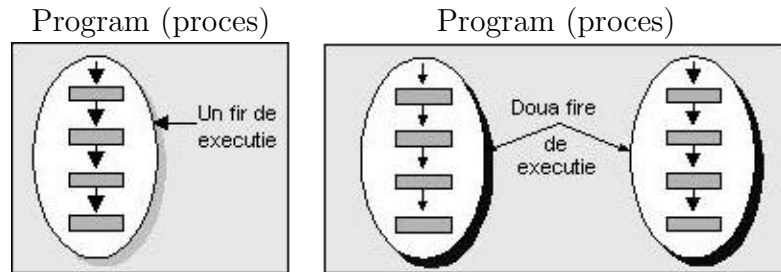
12.1 Introducere

Firele de execuție fac trecerea de la programarea secvențială la programarea concurentă. Un program secvențial reprezintă modelul clasic de program: are un început, o secvență de execuție a instrucțiunilor sale și un sfârșit. Cu alte cuvinte, la un moment dat programul are un singur punct de execuție. Un program aflat în execuție se numește *proces*. Un sistem de operare monotasking, cum ar fi MS-DOS, nu este capabil să execute decât un singur proces la un moment dat, în timp ce un sistem de operare multitasking, cum ar fi UNIX sau Windows, poate rula oricâte procese în același timp (concurent), folosind diverse strategii de alocare a procesorului fiecăruia dintre acestea. Am reamintit acest lucru deoarece noțiunea de fir de execuție nu are sens decât în cadrul unui sistem de operare multitasking.

Un fir de execuție este similar unui proces secvențial, în sensul că are un început, o secvență de execuție și un sfârșit. Diferența dintre un fir de execuție și un proces constă în faptul că un fir de execuție nu poate rula independent ci trebuie să ruleze în cadrul unui proces.

Definiție

Un *fir de execuție* este o succesiune secvențială de instrucțiuni care se execută în cadrul unui proces.



Un program își poate defini însă nu doar un fir de execuție ci oricâte, ceea ce înseamnă că în cadrul unui proces se pot executa simultan mai multe fire de execuție, permițând efectuarea concurrentă a sarcinilor independente ale aceluiași program.

Un fir de execuție poate fi asemănat cu o versiune redusă a unui proces, ambele rulând simultan și independent pe o structură secvențială formată de instrucțiunile lor. De asemenea, execuția simultană a firelor în cadrul unui proces este similară cu execuția concurrentă a proceselor: sistemul de operare va aloca procesorul după o anumită strategie fiecărui fir de execuție până la terminarea lor. Din acest motiv firele de execuție mai sunt numite și *procese usoare*.

Care ar fi însă deosebirile între un fir de execuție și un proces? În primul rând deosebirea majoră constă în faptul că firele de execuție nu pot rula decât în cadrul unui proces. O altă deosebire rezultă din faptul că fiecare proces are propria sa memorie (propriul său spațiu de adrese) iar la crearea unui nou proces (fork) este realizată o copie exactă a procesului părinte: cod și date, în timp ce la crearea unui fir nu este copiat decât codul procesului părinte, toate firele de execuție având acces la aceleași date, datele procesului original. Așadar, un fir mai poate fi privit și ca un *context de execuție* în cadrul unui proces.

Firele de execuție sunt utile în multe privințe, însă uzual ele sunt folosite pentru executarea unor operații consumatoare de timp fără a bloca procesul principal: calcule matematice, așteptarea eliberării unei resurse, desenarea componentelor unei aplicații GUI, etc. De multe ori ori, firele își desfășoară activitatea în fundal însă, evident, acest lucru nu este obligatoriu.

12.2 Crearea unui fir de execuție

Ca orice alt obiect Java, un fir de execuție este o instanță a unei clase. Firele de execuție definite de o clasă vor avea același cod și, prin urmare, aceeași

secvența de instrucțiuni. Crearea unei clase care să definească fire de execuție poate fi făcută prin două modalități:

- prin extinderea clasei **Thread**.
- prin implementarea interfeței **Runnable**.

Orice clasă ale cărei instanțe vor fi executate separat într-un fir propriu trebuie declarată ca fiind de tip **Runnable**. Aceasta este o interfață care conține o singură metodă și anume metoda **run**. Așadar, orice clasă ce descrie fire de execuție va conține metoda **run** în care este implementat codul ce va fi rulat. Interfața **Runnable** este concepută ca fiind un protocol comun pentru obiectele care doresc să execute un anumit cod pe durata existenței lor.

Cea mai importantă clasă care implementează interfața **Runnable** este **Thread**. Aceasta implementează un fir de execuție generic care, implicit, nu face nimic; cu alte cuvinte, metoda **run** nu conține nici un cod. Orice fir de execuție este o instanță a clasei **Thread** sau a unei subclase a sa.

12.2.1 Extinderea clasei **Thread**

Cea mai simplă metodă de a crea un fir de execuție care să realizeze o anumită acțiune este prin extinderea clasei **Thread** și supradefinirea metodei **run** a acesteia. Formatul general al unei astfel de clase este:

```
public class FirExcecutie extends Thread {  
  
    public FirExcecutie(String nume) {  
        // Apelam constructorul superclasei  
        super(nume);  
    }  
  
    public void run() {  
        // Codul firului de executie  
        ...  
    }  
}
```

Prima metodă a clasei este constructorul, care primește ca argument un șir ce va reprezenta numele firului de execuție. În cazul în care nu vrem să dăm nume firelor pe care le creăm, atunci putem renunța la supradefinirea

acestui constructor și să folosim constructorul implicit, fără argumente, care creează un fir de execuție fără nici un nume. Ulterior, acesta poate primi un nume cu metoda `setName`. Evident, se pot defini și alți constructori, aceștia fiind utili atunci când vrem să trimitem diverși parametri de inițializare firului nostru. A doua metodă este metoda `run`, ”inima” oricărui fir de execuție, în care scriem efectiv codul care trebuie să se execute.

Un fir de execuție creat nu este automat pornit, lansarea să fiind realizează de metoda `start`, definită în clasa `Thread`.

```
// Cream firul de executie
FirExecutie fir = new FirExecutie("simplu");

// Lansam in executie
fir.start();
```

Să considerăm în continuare un exemplu în care definim un fir de execuție ce afișează numerele întregi dintr-un interval, cu un anumit pas.

Listing 12.1: Folosirea clasei `Thread`

```
class AfisareNumere extends Thread {
    private int a, b, pas;

    public AfisareNumere(int a, int b, int pas) {
        this.a = a;
        this.b = b;
        this.pas = pas;
    }

    public void run() {
        for(int i = a; i <= b; i += pas)
            System.out.print(i + " ");
    }
}

public class TestThread {
    public static void main(String args[]) {
        AfisareNumere fir1, fir2;

        fir1 = new AfisareNumere(0, 100, 5);
        // Numara de la 0 la 100 cu pasul 5

        fir2 = new AfisareNumere(100, 200, 10);
```

```

// Numara de la 100 la 200 cu pasul 10

fir1.start();
fir2.start();
// Pornim firele de executie
// Ele vor fi distruse automat la terminarea lor
}
}

```

Gândind secvențial, s-ar crede că acest program va afișa prima dată numerele de la 0 la 100 cu pasul 5, apoi numerele de la 100 la 200 cu pasul 10, întrucât primul apel este către contorul `fir1`, deci rezultatul afișat pe ecran ar trebui să fie:

```

0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100
100 110 120 130 140 150 160 170 180 190 200

```

În realitate însă, rezultatul obținut va fi o intercalare de valori produse de cele două fire ce rulează simultan. La rulări diferite se pot obține rezultate diferite deoarece timpul alocat fiecărui fir de execuție poate să nu fie același, el fiind controlat de procesor într-o manieră "aparent" aleatoare. Un posibil rezultat al programului de mai sus:

```

0 100 5 110 10 120 15 130 20 140 25 150 160 170 180 190 200
30 35 40 45 50 55 60 65 70 75 80 85 90 95 100

```

12.2.2 Implementarea interfeței Runnable

Ce facem însă când dorim să creăm o clasă care instanțiază fire de execuție dar aceasta are deja o superclasă, știind că în Java nu este permisă moștenirea multiplă ?

```

class FirExecutie extends Parinte, Thread // incorect !

```

În acest caz, nu mai putem extinde clasa `Thread` ci trebuie să implementăm direct interfața `Runnable`. Clasa `Thread` implementează ea însăși interfața `Runnable` și, din acest motiv, la extinderea ei obținem implementarea indirectă a interfeței. Așadar, interfața `Runnable` permite unei clase să fie activă, fără a extinde clasa `Thread`.

Interfața `Runnable` se găsește în pachetul `java.lang` și este definită astfel:

```
public interface Runnable {
    public abstract void run();
}
```

Prin urmare, o clasă care instanțiază fire de execuție prin implementarea interfeței `Runnable` trebuie obligatoriu să implementeze metoda `run`. O astfel de clasă se mai numește *clasă activă* și are următoarea structură:

```
public class ClasaActiva implements Runnable {

    public void run() {
        //Codul firului de executie
        ...
    }
}
```

Spre deosebire de modalitatea anterioară, se pierde însă tot suportul oferit de clasa `Thread`. Simpla instanțiere a unei clase care implementează interfața `Runnable` nu creează nici un fir de execuție, crearea acestora trebuind făcută explicit. Pentru a realiza acest lucru trebuie să instanțiem un obiect de tip `Thread` ce va reprezenta firul de execuție propriu zis al cărui cod se găsește în clasa noastră. Acest lucru se realizează, ca pentru orice alt obiect, prin instrucțiunea `new`, urmată de un apel la un constructor al clasei `Thread`, însă nu la oricare dintre aceștia. Trebuie apelat constructorul care să primească drept argument o instanță a clasei noastre. După creare, firul de execuție poate fi lansat printr-un apel al metodei `start`.

```
ClasaActiva obiectActiv = new ClasaActiva();
Thread fir = new Thread(obiectActiv);
fir.start();
```

Aceste operațiuni pot fi făcute chiar în cadrul clasei noastre:

```
public class FirExecutie implements Runnable {

    private Thread fir = null;

    public FirExecutie()
        if (fir == null) {
            fir = new Thread(this);
        }
}
```

```
        fir.start();
    }
}

public void run() {
    //Codul firului de executie
    ...
}
}
```

Specificarea argumentului `this` în constructorul clasei `Thread` determină crearea unui fir de execuție care, la lansarea sa, va apela metoda `run` din clasa curentă. Așadar, acest constructor acceptă ca argument orice instanță a unei clase "Runnable". Pentru clasa `FirExecutie` dată mai sus, lansarea firului va fi făcută automat la instanțierea unui obiect al clasei:

```
FirExecutie fir = new FirExecutie();
```

Atenție

Metoda `run` nu trebuie apelată explicit, acest lucru realizându-se automat la apelul metodei `start`. Apelul explicit al metodei `run` nu va furniza nici o eroare, însă aceasta va fi executată ca orice altă metoda, și nu separat într-un fir.

Să considerăm următorul exemplu în care creăm două fire de execuție folosind interfața `Runnable`. Fiecare fir va desena figuri geometrice de un anumit tip, pe o suprafață de desenare de tip `Canvas`. Vom porni apoi două fire de execuție care vor rula concurent, desenând figuri diferite, fiecare pe suprafața sa.

Listing 12.2: Folosirea interfeței `Runnable`

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

class Plansa extends Canvas implements Runnable {
    // Deoarece Plansa extinde Canvas,
```

```

// nu mai putem extinde clasa Thread

Dimension dim = new Dimension(300, 300);
Color culoare;
String figura;
int x=0, y=0, r=0;

public Plansa(String figura, Color culoare) {
    this.figura = figura;
    this.culoare = culoare;
}

public Dimension getPreferredSize() {
    return dim;
}

public void paint(Graphics g) {
    // Desenam un chenar
    g.setColor(Color.black);
    g.drawRect(0, 0, dim.width-1, dim.height-1);

    // Desenam figura la coordonatele calculate
    // de firul de executie
    g.setColor(culoare);
    if (figura.equals("patrat"))
        g.drawRect(x, y, r, r);
    else
        if (figura.equals("cerc"))
            g.drawOval(x, y, r, r);
}

public void update(Graphics g) {
    paint(g);
    // Supradefinim update ca sa nu mai
    // fie stearsa suprafata de desenare
}

public void run() {
    /* Codul firului de executie:
       Afisarea a 100 de figuri geometrice la pozitii
       si dimensiuni calculate aleator.
       Intre doua afisari, facem o pauza de 50 ms
    */

    for(int i=0; i<100; i++) {

```

```
        x = (int)(Math.random() * dim.width);
        y = (int)(Math.random() * dim.height);
        r = (int)(Math.random() * 100);
        try {
            Thread.sleep(50);
        } catch (InterruptedException e) {}
        repaint();
    }
}

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

        // Cream doua obiecte active de tip Plansa
        Plansa p1 = new Plansa("patrat", Color.blue);
        Plansa p2 = new Plansa("cerc", Color.red);

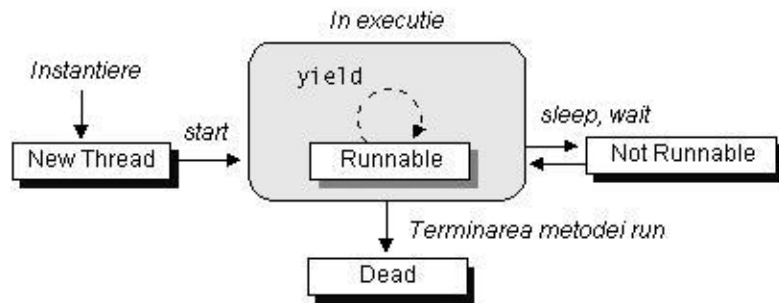
        // Acestea extind Canvas, le plasam pe fereastra
        setLayout(new GridLayout(1, 2));
        add(p1);
        add(p2);
        pack();

        // Pornim doua fire de executie, care vor
        // actualiza desenul celor doua planse
        new Thread(p1).start();
        new Thread(p2).start();
    }
}

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

12.3 Ciclul de viață al unui fir de execuție

Fiecare fir de execuție are propriul său ciclu de viață: este creat, devine activ prin lansarea sa și, la un moment dat, se termină. În continuare, vom analiza mai îndeaproape stările în care se poate găsi un fir de execuție. Diagrama de mai jos ilustrează generic aceste stări precum și metodele care provoacă tranziția dintr-o stare în alta:



Așadar, un fir de execuție se poate găsi în una din următoarele patru stări:

- "New Thread"
- "Runnable"
- "Not Runnable"
- "Dead"

Starea "New Thread"

Un fir de execuție se găsește în această stare imediat după crearea sa, cu alte cuvinte după instanțierea unui obiect din clasa `Thread` sau dintr-o subclasă a sa.

```

Thread fir = new Thread(obiectActiv);
// fir se gaseste in starea "New Thread"
  
```

În această stare firul este "vid", el nu are alocate nici un fel de resurse sistem și singura operațiune pe care o putem executa asupra lui este lansarea în execuție, prin metoda `start`. Apelul oricărei alte metode în afară de `start` nu are nici un sens și va provoca o excepție de tipul `IllegalThreadStateException`.

Starea "Runnable"

După apelul metodei `start` un fir va trece în starea "Runnable", adică va fi în execuție.

```
fir.start();  
//fir se gaseste in starea "Runnable"
```

Metoda `start` realizează următoarele operațiuni necesare rulării firului de execuție:

- Alocă resursele sistem necesare.
- Planifică firul de execuție la procesor pentru a fi lansat.
- Apelează metoda `run` a obiectului activ al firului.

Un fir aflat în starea "Runnable" nu înseamnă neapărat că se găsește efectiv în execuție, adică instrucțiunile sale sunt interpretate de procesor. Acest lucru se întâmplă din cauza că majoritatea calculatoarelor au un singur procesor iar acesta nu poate rula simultan toate firele de execuție care se găsesc în starea "Runnable". Pentru a rezolva aceasta problemă există o planificare care să partajeze dinamic și corect procesorul între toate firele de execuție care sunt în starea "Runnable". Așadar, un fir care "rulează" poate să-și aștepte de fapt rândul la procesor.

Starea "Not Runnable"

Un fir de execuție poate ajunge în această stare în una din următoarele situații:

- Este "adormit" prin apelul metodei `sleep`;
- A apelat metoda `wait`, așteptând ca o anumită condiție să fie satisfăcută;

- Este blocat într-o operație de intrare/ieșire.

Metoda `sleep` este o metodă statică a clasei `Thread` care provoacă o pauză în timpul rulării firului curent aflat în execuție, cu alte cuvinte îl "adoarme" pentru un timp specificat. Lungimea acestei pauze este specificată în milisecunde și chiar nanosecunde. Întrucât poate provoca excepții de tipul `InterruptedException`, apelul acestei metode se face într-un bloc de tip `try-catch`:

```
try {
    // Facem pauza de o secunda
    Thread.sleep(1000);
} catch (InterruptedException e) {
    ...
}
```

Observați că metoda fiind statică apelul ei nu se face pentru o instanță anume a clasei `Thread`. Acest lucru este foarte normal deoarece, la un moment dat, un singur fir este în execuție și doar pentru acesta are sens "adormirea" sa.

În intervalul în care un fir de execuție "doarme", acesta nu va fi executat chiar dacă procesorul devine disponibil. După expirarea intervalului specificat firul revine în starea "Runnable" iar dacă procesorul este în continuare disponibil își va continua execuția.

Pentru fiecare tip de intrare în starea "Not Runnable", există o secvență specifică de ieșire din starea respectivă, care readuce firul de execuție în starea "Runnable". Acestea sunt:

- Dacă un fir de execuție a fost "adormit", atunci el devine "Runnable" doar după scurgerea intervalului de timp specificat de instrucțiunea `sleep`.
- Dacă un fir de execuție așteaptă o anumită condiție, atunci un alt obiect trebuie să îl informeze dacă acea condiție este îndeplinită sau nu; acest lucru se realizează prin instrucțiunile `notify` sau `notifyAll` (vezi "Sincronizarea firelor de execuție").
- Dacă un fir de execuție este blocat într-o operațiune de intrare/ieșire atunci el redevine "Runnable" atunci când acea operațiune s-a terminat.

Starea "Dead"

Este starea în care ajunge un fir de execuție la terminarea sa. Un fir nu poate fi oprit din program printr-o anumită metodă, ci trebuie să se termine în mod natural la încheierea metodei `run` pe care o execută. Spre deosebire de versiunile curente ale limbajului Java, în versiunile mai vechi exista metoda `stop` a clasei `Thread` care termina forțat un fir de execuție, însă aceasta a fost eliminată din motive de securitate. Așadar, un fir de execuție trebuie să-și "aranjeze" singur propria sa "moarte".

12.3.1 Terminarea unui fir de execuție

După cum am văzut, un fir de execuție nu poate fi terminat forțat de către program ci trebuie să-și "aranjeze" singur terminarea sa. Acest lucru poate fi realizat în două modalități:

1. Prin scrierea unor metode `run` care să-și termine execuția în mod natural. La terminarea metodei `run` se va termina automat și firul de execuție, acesta intrând în starea `Dead`. Ambele exemple anteriorare se încadrează în această categorie.

```
// Primul exemplu
public void run() {
    for(int i = a; i <= b; i += pas)
        System.out.print(i + " ");
}
```

După afișarea numerelor din intervalul specificat, metoda se termină și, odată cu ea, se va termina și firul de execuție respectiv.

2. Prin folosirea unei variabile de terminare. În cazul când metoda `run` trebuie să execute o buclă infinită atunci aceasta trebuie controlată printr-o variabilă care să oprească ciclul atunci când dorim ca firul de execuție să se termine. Uzual, vom folosi o variabilă membră a clasei care descrie firul de execuție care fie este publică, fie este asociată cu o metodă publică care permite schimbarea valorii sale.

Să considerăm exemplul unui fir de execuție care trebuie să numere secunde scurse până la apăsarea tastei `Enter`.

Listing 12.3: Folosirea unei variabile de terminare

```
import java.io.*;

class NumaraSecunde extends Thread {
    public int sec = 0;
    // Folosim o variabila de terminare
    public boolean executie = true;

    public void run() {
        while (executie) {
            try {
                Thread.sleep(1000);
                sec++;
                System.out.print(".");
            } catch (InterruptedException e){}
        }
    }
}

public class TestTerminare {
    public static void main(String args[])
        throws IOException {

        NumaraSecunde fir = new NumaraSecunde();
        fir.start();

        System.out.println("Apasati tasta Enter");
        System.in.read();

        // Oprim firul de executie
        fir.executie = false;
        System.out.println("S-au scurs " + fir.sec + " secunde");
    }
}
```

Nu este necesară distrugerea explicită a unui fir de execuție. Sistemul Java de colectare a "gunoiului" se ocupă de acest lucru. Setarea valorii null pentru variabila care referea instanța firului de execuție va ușura însă activitatea procesului *gc*.

Metoda `System.exit` va oprit forțat toate firele de execuție și va termina aplicația curentă.

Pentru a testa dacă un fir de execuție a fost pornit dar nu s-a terminat încă putem folosi metoda **isAlive**. Metoda returnează:

- **true** - dacă firul este în una din stările "Runnable" sau "Not Runnable"
- **false** - dacă firul este în una din stările "New Thread" sau "Dead"

Intre stările "Runnable" sau "Not Runnable", respectiv "New Thread" sau "Dead" nu se poate face nici o diferențiere.

```
NumaraSecunde fir = new NumaraSecunde();
// isAlive returneaza false (starea este New Thread)

fir.start();
// isAlive returneaza true (starea este Runnable)

fir.executie = false;
// isAlive returneaza false (starea este Dead)
```

12.3.2 Fire de execuție de tip "daemon"

Un proces este considerat în execuție dacă conține cel puțin un fir de execuție activ. Cu alte cuvinte, la rularea unei aplicații, mașina virtuală Java nu se va opri decât atunci când nu mai există nici un fir de execuție activ. De multe ori însă dorim să folosim fire care să realizeze diverse activități, eventual periodic, pe toată durata de execuție a programului iar în momentul terminării acestuia să se termine automat și firele respective. Aceste fire de execuție se numesc *demoni*.

După crearea sa, un fir de execuție poate fi făcut demon, sau scos din această stare, cu metoda **setDaemon**.

Listing 12.4: Crearea unui fir de execuție de tip "daemon"

```
class Beeper implements Runnable {
    public void run() {
        while (true) {
            java.awt.Toolkit.getDefaultToolkit().beep();
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {}
        }
    }
}
```

```

    }
}

public class TestDaemon {
    public static void main(String args[])
        throws java.io.IOException {

        Thread t = new Thread(new Beeper());
        t.setDaemon(true);
        t.start();

        System.out.println("Apasati Enter...");
        System.in.read();

        // "Demonul" se termina automat
        // la terminarea aplicatiei
    }
}

```

12.3.3 Stabilirea priorităților de execuție

Majoritatea calculatoarelor au un sigur procesor, ceea ce înseamnă că firele de execuție trebuie să-și împartă accesul la acel procesor. Execuția într-o anumită ordine a mai multor fire de execuție pe un număr limitat de procesoare se numește *planificare (scheduling)*. Sistemul Java de execuție a programelor implementează un algoritm simplu, determinist de planificare, cunoscut sub numele de *planificare cu priorități fixate*.

Fiecare fir de execuție Java primește la crearea sa o anumită prioritate. O prioritate este de fapt un număr întreg cu valori cuprinse între `MIN_PRIORITY` și `MAX_PRIORITY`. Implicit, prioritatea unui fir nou creat are valoarea `NORM_PRIORITY`. Aceste trei constante sunt definite în clasa `Thread` astfel:

```

public static final int MAX_PRIORITY = 10;
public static final int MIN_PRIORITY = 1;
public static final int NORM_PRIORITY= 5;

```

Schimbarea ulterioară a priorității unui fir de execuție se realizează cu metoda `setPriority` a clasei `Thread`.

La nivelul sistemului de operare, există două modele de lucru cu fire de execuție:

- *Modelul cooperativ*, în care firele de execuție decid când să cedeze procesorul; dezavantajul acestui model este că unele fire pot acapara procesorul, nepermițând și execuția altora până la terminarea lor.
- *Modelul preemptiv*, în care firele de execuție pot fi întrerupte oricând, după ce au fost lăsate să ruleze o perioadă, urmând să fie reluate după ce și celelalte fire aflate în execuție au avut acces la procesor; acest sistem se mai numește cu "cuante de timp", dezavantajul său fiind nevoia de a sincroniza accesul firelor la resursele comune.

Așadar, în modelul cooperativ firele de execuție sunt responsabile cu partajarea timpului de execuție, în timp ce în modelul preemptiv ele trebuie să partajeze resursele comune. Deoarece specificațiile mașinii virtuale Java nu impun folosirea unui anumit model, programele Java trebuie scrise astfel încât să funcționeze corect pe ambele modele. În continuare, vom mai detalia puțin aceste aspecte.

Planificatorul Java lucrează în modul următor: dacă la un moment dat sunt mai multe fire de execuție în starea "Runnable", adică sunt pregătite pentru a fi rulate, planificatorul îl va alege pe cel cu prioritatea cea mai mare pentru a-l executa. Doar când firul de execuție cu prioritate maximă se termină, sau este suspendat din diverse motive, va fi ales un fir cu o prioritate mai mică. În cazul în care toate firele au aceeași prioritate ele sunt alese pe rând, după un algoritm simplu de tip "round-robin". De asemenea, dacă un fir cu prioritate mai mare decât firul care se execută la un moment dat solicită procesorul, atunci firul cu prioritate mai mare este imediat trecut în execuție iar celalalt trecut în așteptare. Planificatorul Java nu va întrerupe însă un fir de execuție în favoarea altuia de aceeași prioritate, însă acest lucru îl poate face sistemul de operare în cazul în care acesta alocă procesorul în cuante de timp (un astfel de SO este Windows).

Așadar, un fir de execuție Java cedează procesorul în una din situațiile:

- un fir de execuție cu o prioritate mai mare solicită procesorul;
 - metoda sa `run` se termină;
 - face explicit acest lucru apelând metoda `yield`;
 - timpul alocat pentru execuția sa a expirat (pe SO cu cuante de timp).
-

Atenție

În nici un caz corectitudinea unui program nu trebuie să se bazeze pe mecanismul de planificare a firelor de execuție, deoarece acesta poate fi diferit de la un sistem de operare la altul.

Un fir de execuție de lungă durată și care nu cedează explicit procesorul la anumite intervale de timp astfel încât să poată fi executate și celelalte fire de execuție se numește fir de execuție *egoist*. Evident, trebuie evitată scrierea lor întrucât acaparează pe termen nedefinit procesorul, blocând efectiv execuția celorlalte fire de execuție până la terminarea sa. Unele sistemele de operare combat acest tip de comportament prin metoda alocării procesorului în cuante de timp fiecărui fir de execuție, însă nu trebuie să ne bazăm pe acest lucru la scrierea unui program. Un fir de execuție trebuie să fie "corect" față de celelalte fire și să cedeze periodic procesorul astfel încât toate să aibă posibilitatea de a se executa.

Listing 12.5: Exemplu de fir de execuție "egoist"

```
class FirEgoist extends Thread {
    public FirEgoist(String name) {
        super(name);
    }
    public void run() {
        int i = 0;
        while (i < 100000) {
            // Bucla care acapareaza procesorul
            i ++;
            if (i % 100 == 0)
                System.out.println(getName() + " a ajuns la " + i);
            // yield();
        }
    }
}

public class TestFirEgoist {
    public static void main(String args[]) {
        FirEgoist s1, s2;
        s1 = new FirEgoist("Firul 1");
        s1.setPriority (Thread.MAX_PRIORITY);
        s2 = new FirEgoist("Firul 2");
        s2.setPriority (Thread.MAX_PRIORITY);
    }
}
```

```
s1.start();
s2.start();
}
}
```

Firul de execuție *s1* are prioritate maximă și până nu-și va termina execuția nu-i va permite firului *s2* să execute nici o instrucțiune, acaparând efectiv procesorul. Rezultatul va arăta astfel:

```
Firul 1 a ajuns la 100
Firul 1 a ajuns la 200
Firul 1 a ajuns la 300
...
Firul 1 a ajuns la 99900
Firul 1 a ajuns la 100000
Firul 2 a ajuns la 100
Firul 2 a ajuns la 200
...
Firul 2 a ajuns la 99900
Firul 2 a ajuns la 100000
```

Rezolvarea acestei probleme se face fie prin intermediul metodei statice `yield` a clasei `Thread`, care determină firul de execuție curent să se oprească temporar, dând ocazia și altor fire să se execute, fie prin "adormirea" temporară a firului curent cu ajutorul metodei `sleep`. Prin metoda `yield` un fir de execuție nu cedează procesorul decât firelor de execuție care au aceeași prioritate cu a sa și nu celor cu priorități mai mici. Decomentând linia în care apelăm `yield` din exemplul anterior, execuția celor două fire se va intercala.

```
...
Firul 1 a ajuns la 31900
Firul 1 a ajuns la 32000
Firul 2 a ajuns la 100
Firul 1 a ajuns la 32100
Firul 2 a ajuns la 200
Firul 2 a ajuns la 300
...
```

12.3.4 Sincronizarea firelor de execuție

Până acum am văzut cum putem crea fire de execuție independente și asincrone, cu alte cuvinte care nu depind în nici un fel de execuția sau de rezultatele altor fire. Există însă numeroase situații când fire de execuție separate, dar care rulează concurrent, trebuie să comunice între ele pentru a accesa diferite resurse comune sau pentru a-și transmite dinamic rezultatele "muncii" lor. Cel mai elocvent scenariu în care firele de execuție trebuie să se comunice între ele este cunoscut sub numele de problema *producătorului/consumatorului*, în care producătorul generează un flux de date care este preluat și prelucrat de către consumator.

Să considerăm de exemplu o aplicație Java în care un fir de execuție (producătorul) scrie date într-un fișier în timp ce alt fir de execuție (consumatorul) citește date din același fișier pentru a le prelucra. Sau, să presupunem că producătorul generează niște numere și le plasează, pe rând, într-un buffer iar consumatorul citește numerele din acel buffer pentru a le procesa. În ambele cazuri avem de-a face cu fire de execuție concurente care folosesc o resursă comună: un fișier, respectiv o zonă de memorie și, din acest motiv, ele trebuie sincronizate într-o manieră care să permită decurgerea normală a activității lor.

12.3.5 Scenariul producător / consumator

Pentru a înțelege mai bine modalitatea de sincronizare a două fire de execuție să implementăm efectiv o problemă de tip producător/consumator. Să considerăm următoarea situație:

- *Producătorul* generează numerele întregi de la 1 la 10, fiecare la un interval neregulat cuprins între 0 și 100 de milisecunde. Pe măsura ce le generează încearcă să le plaseze într-o zonă de memorie (o variabilă întreaga) de unde să fie citite de către consumator.
- *Consumatorul* va prelua, pe rând, numerele generate de către producător și va afișa valoarea lor pe ecran.

Pentru a fi accesibilă ambelor fire de execuție, vom încapsula variabila ce va conține numerele generate într-un obiect descris de clasa **Buffer** și care va avea două metode **put** (pentru punerea unui număr în buffer) și **get** (pentru obținerea numărului din buffer).

Fără a folosi nici un mecanism de sincronizare clasa `Buffer` arată astfel:

Listing 12.6: Clasa `Buffer` fără sincronizare

```
class Buffer {
    private int number = -1;

    public int get() {
        return number;
    }

    public void put(int number) {
        this.number = number;
    }
}
```

Vom implementa acum clasele `Producator` și `Consumator` care vor descrie cele două fire de execuție. Ambele vor avea o referința comună la un obiect de tip `Buffer` prin intermediul căruia își comunică valorile.

Listing 12.7: Clasele `Producator` și `Consumator`

```
class Producator extends Thread {
    private Buffer buffer;

    public Producator(Buffer b) {
        buffer = b;
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            buffer.put(i);
            System.out.println("Producatorul a pus:\t" + i);
            try {
                sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) { }
        }
    }
}

class Consumator extends Thread {
    private Buffer buffer;

    public Consumator(Buffer b) {
        buffer = b;
    }
}
```

```
    }

    public void run() {
        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = buffer.get();
            System.out.println("Consumatorul a primit:\t" + value);
        }
    }
}

public class TestSincronizare1 {
    public static void main(String[] args) {
        Buffer b = new Buffer();
        Producator p1 = new Producator(b);
        Consumator c1 = new Consumator(b);
        p1.start();
        c1.start();
    }
}
```

După cum ne așteptam, rezultatul rulării acestui program nu va rezolva nici pe departe problema propusă de noi, motivul fiind lipsa oricărei sincronizări între cele două fire de execuție. Mai precis, rezultatul va fi ceva de forma:

```
Consumatorul a primit: -1
Consumatorul a primit: -1
Producatorul a pus:    0
Consumatorul a primit: 0
Consumatorul a primit: 0
Consumatorul a primit: 0
Consumatorul a primit: 0
Consumatorul a primit: 0
Consumatorul a primit: 0
Consumatorul a primit: 0
Consumatorul a primit: 0
Consumatorul a primit: 0
Consumatorul a primit: 0
Consumatorul a primit: 0
Producatorul a pus:    1
Producatorul a pus:    2
Producatorul a pus:    3
Producatorul a pus:    4
Producatorul a pus:    5
```

```
Producatorul a pus: 6
Producatorul a pus: 7
Producatorul a pus: 8
Producatorul a pus: 9
```

Ambele fire de execuție accesează resursa comună, adică obiectul de tip `Buffer`, într-o manieră haotică și acest lucru se întâmplă din două motive :

- Consumatorul nu așteaptă înainte de a citi ca producătorul să genereze un număr și va prelua de mai multe ori același număr.
- Producătorul nu așteaptă consumatorul să preia numărul generat înainte de a produce un altul, în felul acesta consumatorul va ”rata” cu siguranță unele numere (în cazul nostru aproape pe toate).

Problema care se ridică în acest moment este: cine trebuie să se ocupe de sincronizarea celor două fire de execuție : clasele `Producator` și `Consumator` sau resursa comuna `Buffer` ? Răspunsul este evident: resursa comună `Buffer`, deoarece ea trebuie să permita sau nu accesul la conținutul său și nu firele de execuție care o folosesc. În felul acesta efortul sincronizării este transferat de la producător/consumator la un nivel mai jos, cel al resursei critice.

Activitățile producătorului și consumatorului trebuie sincronizate la nivelul resursei comune în două privințe:

- Cele două fire de execuție nu trebuie să acceseze simultan buffer-ul; acest lucru se realizează prin blocarea obiectului `Buffer` atunci când este accesat de un fir de execuție, astfel încât nici un alt fir de execuție să nu-l mai poată accesa (vezi ”Monitoare”).
- Cele două fire de execuție trebuie să se coordoneze, adică producătorul trebuie să găsească o modalitate de a ”spune” consumatorului că a plasat o valoare în buffer, iar consumatorul trebuie să comunice producătorului că a preluat această valoare, pentru ca acesta să poată genera o alta. Pentru a realiza această comunicare, clasa `Thread` pune la dispoziție metodele `wait`, `notify`, `notifyAll`. (vezi ”Semafoare”).

Folosind sincronizarea clasa `Buffer` va arăta astfel:

Listing 12.8: Clasa Buffer cu sincronizare

```
class Buffer {
    private int number = -1;
    private boolean available = false;

    public synchronized int get() {
        while (!available) {
            try {
                wait();
                // Asteapta producatorul sa puna o valoare
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        available = false;
        notifyAll();
        return number;
    }

    public synchronized void put(int number) {
        while (available) {
            try {
                wait();
                // Asteapta consumatorul sa preia valoarea
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        this.number = number;
        available = true;
        notifyAll();
    }
}
```

Rezultatul obtinut va fi cel scontat:

```
Producatorul a pus:    0
Consumatorul a primit: 0
Producatorul a pus:    1
Consumatorul a primit: 1
...
Producatorul a pus:    9
Consumatorul a primit: 9
```

12.3.6 Monitoare

Definiție

Un segment de cod ce gestionează o resursă comună mai multor de fire de execuție separate concurente se numește *secțiune critică*. În Java, o secțiune critică poate fi un bloc de instrucțiuni sau o metodă.

Controlul accesului într-o secțiune critică se face prin cuvântul cheie **synchronized**. Platforma Java asociază un monitor ("lacăt") fiecărui obiect al unui program aflat în execuție. Acest monitor va indica dacă resursa critică este accesată de vreun fir de execuție sau este liberă, cu alte cuvinte "monitorizează" resursa respectivă. În cazul în care este accesată, va pune un lacăt pe aceasta, astfel încât să împiedice accesul altor fire de execuție la ea. În momentul când resursa este eliberată "lacătul" va fi eliminat, pentru a permite accesul altor fire de execuție.

În exemplul de tip producător/consumator de mai sus, secțiunile critice sunt metodele **put** și **get** iar resursa critică comună este obiectul **buffer**. Consumatorul nu trebuie să acceseze buffer-ul când producatorul tocmai pune o valoare în el, iar producătorul nu trebuie să modifice valoarea din buffer în momentul când aceasta este citită de către consumator.

```
public synchronized int get() {  
    ...  
}  
public synchronized void put(int number) {  
    ...  
}
```

Să observăm că ambele metode au fost declarate cu modificatorul **synchronized**. Cu toate acestea, sistemul asociază un monitor unei instanțe a clasei **Buffer** și nu unei metode anume. În momentul în care este apelată o metodă sincronizată, firul de execuție care a făcut apelul va bloca obiectul a cărei metodă o accesează, ceea ce înseamnă că celelalte fire de execuție nu vor mai putea accesa resursele critice ale aceluși obiect. Acesta este un lucru logic, deoarece mai multe secțiuni critice ale unui obiect gestionează de fapt o singură resursă critică.

În exemplul nostru, atunci când producătorul apelează metoda **put** pentru a scrie un număr, va bloca tot obiectul **buffer**, astfel că firul de execuție

consumator nu va avea acces la metoda `get`, și reciproc.

```
public synchronized void put(int number) {
    // buffer blocat de producator
    ...
    // buffer deblocat de producator
}
public synchronized int get() {
    // buffer blocat de consumator
    ...
    // buffer deblocat de consumator
}
```

Monitoare fine

Adeseori, folosirea unui monitor pentru întreg obiectul poate fi prea restrictivă. De ce să blocăm toate resursele unui obiect dacă un fir de execuție nu dorește decât accesarea uneia sau a câtorva dintre ele ? Deoarece orice obiect are un monitor, putem folosi obiecte fictive ca lacăte pentru fiecare din resursele obiectului nostru, ca în exemplul de mai jos:

```
class MonitoareFine {
    //Cele doua resurse ale obiectului
    Resursa x, y;

    //Folosim monitoarele a doua obiecte fictive
    Object xLacat = new Object(),
           yLacat = new Object();

    public void metoda() {
        synchronized(xLacat) {
            // Accesam resursa x
        }
        // Cod care nu foloseste resursele comune
        ...
        synchronized(yLacat) {
            // Accesam resursa y
        }
    }
}
```

```

...
synchronized(xLacat) {
    synchronized(yLacat) {
        // Accesam x si y
    }
}
...
synchronized(this) {
    // Accesam x si y
}
}
}

```

Metoda de mai sus nu a fost declarată cu `synchronized` ceea ce ar fi determinat blocarea tuturor resurselor comune la accesarea obiectului respectiv de un fir de execuție, ci au fost folosite monitoarele unor obiecte fictive pentru a controla folosirea fiecărei resursă în parte.

12.3.7 Semafoare

Obiectul de tip `Buffer` din exemplul anterior are o variabilă membră privată numită `number`, în care este memorat numărul pe care îl comunică producătorul și pe care îl preia consumatorul. De asemenea, mai are o variabilă privată logică `available` care ne dă starea buffer-ului: dacă are valoarea `true` înseamnă că producătorul a pus o valoare în buffer și consumatorul nu a preluat-o încă; dacă este `false`, consumatorul a preluat valoarea din buffer dar producătorul nu a pus deocamdată alta la loc. Deci, la prima vedere, metodele clasei `Buffer` ar trebui să arate astfel:

```

public synchronized int get() {
    while (!available) {
        // Nimic - asteptam ca variabila sa devina true
    }
    available = false;
    return number;
}
public synchronized int put(int number) {
    while (available) {

```

```

    // Nimic - asteptam ca variabila sa devina false
}
available = true;
this.number = number;
}

```

Varianta de mai sus, deși pare corectă, nu este. Aceasta deoarece implementarea metodelor este "selfish", cele două metode își asteaptă în mod egoist condiția de terminare. Ca urmare, corectitudinea funcționării va depinde de sistemul de operare pe care programul este rulat, ceea ce reprezintă o greșeală de programare.

Punerea corectă a unui fir de execuție în așteptare se realizează cu metoda **wait** a clasei **Thread**, care are următoarele forme:

```

void wait( )
void wait( long timeout )
void wait( long timeout, long nanos )

```

După apelul metodei **wait**, firul de execuție curent eliberează monitorul asociat obiectului respectiv și așteaptă ca una din următoarele condiții să fie îndeplinită:

- Un alt fir de execuție informează pe cei care "așteaptă" la un anumit monitor să se "trezească" - acest lucru se realizează printr-un apel al metodei **notifyAll** sau **notify**.
- Perioada de așteptare specificată a expirat.

Metoda **wait** poate produce excepții de tipul **InterruptedException**, atunci când firul de execuție care așteaptă (este deci în starea "Not Runnable") este întrerupt din așteptare și trecut forțat în starea "Runnable", deși condiția așteptată nu era încă îndeplinită.

Metoda **notifyAll** informează toate firele de execuție care sunt în așteptare la monitorul obiectului curent îndeplinirea condiției pe care o așteptau. Metoda **notify** informează doar un singur fir de execuție, specificat ca argument.

Reamintim varianta corectă a clasei **Buffer**:

Listing 12.9: Folosirea semafoarelor

```

class Buffer {
    private int number = -1;

```

```
private boolean available = false;

public synchronized int get() {
    while (!available) {
        try {
            wait();
            // Asteapta producatorul sa puna o valoare
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    available = false;
    notifyAll();
    return number;
}

public synchronized void put(int number) {
    while (available) {
        try {
            wait();
            // Asteapta consumatorul sa preia valoarea
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    this.number = number;
    available = true;
    notifyAll();
}
}
```

12.3.8 Probleme legate de sincronizare

Din păcate, folosirea monitoarelor ridică și unele probleme. Să analizăm câteva dintre ele și posibilele lor soluții:

Deadlock

Deadlock-ul este o problemă clasică într-un mediu în care rulează mai multe fire de execuție și constă în faptul că, la un moment dat, întreg procesul se poate bloca deoarece unele fire așteaptă deblocarea unor monitoare care nu se vor debloca niciodată. Există numeroase exemple în acest sens, cea mai cunoscută fiind "Problema filozofilor". Reformulată, să ne imaginăm două persoane "A" și "B" (fire de execuție) care stau la aceeași masă și tre-

buie să folosească în comun cuțitul și furculița (resursele comune) pentru a mânca. Evident, cele două persoane doresc obținerea ambelor resurse. Să presupunem că "A" a oținut cuțitul și "B" furculița. Firul "A" se va bloca în așteptarea eliberării furculiței iar firul "B" se va bloca în așteptarea eliberării cuțitului, ceea ce conduce la starea de "deadlock". Deși acest exemplu este desprins de realitate, există numeroase situații în care fenomenul de "deadlock" se poate manifesta, multe dintre acestea fiind dificil de detectat.

Există câteva reguli ce pot fi aplicate pentru evitarea deadlock-ului:

- Firele de execuție să solicite resursele în aceeași ordine. Această abordare elimină situațiile de așteptare circulară.
- Folosirea unor monitoare care să controleze accesul la un grup de resurse. În cazul nostru, putem folosi un monitor "tacâmuri" care trebuie blocat înainte de a cere furculița sau cuțitul.
- Folosirea unor variabile care să informeze disponibilitatea resurselor fără a bloca monitoarele asociate acestora.
- Cel mai important, conceperea unei arhitecturi a sistemului care să evite pe cât posibil apariția unor potențiale situații de deaslock.

Variabile volatile

Cuvântul cheie *volatile* a fost introdus pentru a controla unele aspecte legate de optimizările efectuate de unele compilatoare. Să considerăm următorul exemplu:

```
class TestVolatile {
    boolean test;
    public void metoda() {
        test = false;
        // *
        if (test) {
            // Aici se poate ajunge...
        }
    }
}
```

Un compilator care optimizează codul, poate decide că variabila *test* fiind setată pe **false**, corpul *if*-ului nu se va executa și să excludă secvența respectivă din rezultatul compilării. Dacă această clasă ar fi însă accesată de mai multe fire de execuție, variabile *test* ar putea fi setată pe **true** de un alt fir, exact între instrucțiunile de atribuire și *if* ale firului curent.

Declararea unei variabile cu modificatorul **volatile** informează compilatorul să nu optimizeze codul în care aceasta apare, previzionând valoarea pe care variabila o are la un moment dat.

Fire de execuție inaccesibile

Uneori firele de execuție sunt blocate din alte motive decât așteptarea la un monitor, cea mai frecventă situație de acest tip fiind operațiunile de intrare/ieșire (IO) blocante. Când acest lucru se întâmplă celelalte fire de execuție trebuie să poată accesa în continuare obiectul. Dar dacă operațiunea IO a fost făcută într-o metodă sincronizată, acest lucru nu mai este posibil, monitorul obiectului fiind blocat de firul care așteaptă de fapt să realizeze operația de intrare/ieșire. Din acest motiv, operațiile IO nu trebuie făcute în metode sincronizate.

12.4 Gruparea firelor de execuție

Gruparea firelor de execuție pune la dispoziție un mecanism pentru manipularea acestora ca un tot și nu individual. De exemplu, putem să pornim sau să suspendăm toate firele dintr-un grup cu un singur apel de metodă. Gruparea firelor de execuție se realizează prin intermediul clasei **ThreadGroup**.

Fiecare fir de execuție Java este membru al unui grup, indiferent dacă specificăm explicit sau nu acest lucru. Afilierea unui fir la un anumit grup se realizează la crearea sa și devine permanentă, în sensul că nu vom putea muta un fir dintr-un grup în altul, după ce acesta a fost creat. În cazul în care creăm un fir folosind un constructor care nu specifică din ce grup face parte, el va fi plasat automat în același grup cu firul de execuție care l-a creat. La pornirea unui program Java se creează automat un obiect de tip **ThreadGroup** cu numele **main**, care va reprezenta grupul tuturor firelor de execuție create direct din program și care nu au fost atașate explicit altui grup. Cu alte cuvinte, putem să ignorăm complet plasarea firelor de execuție în grupuri și să lăsăm sistemul să se ocupe cu aceasta, adunându-le pe toate

în grupul `main`.

Există situații însă când gruparea firelor de execuție poate ușura substanțial manevrarea lor. Crearea unui fir de execuție și plasarea lui într-un grup (altul decât cel implicit) se realizează prin următorii constructori ai clasei `Thread`:

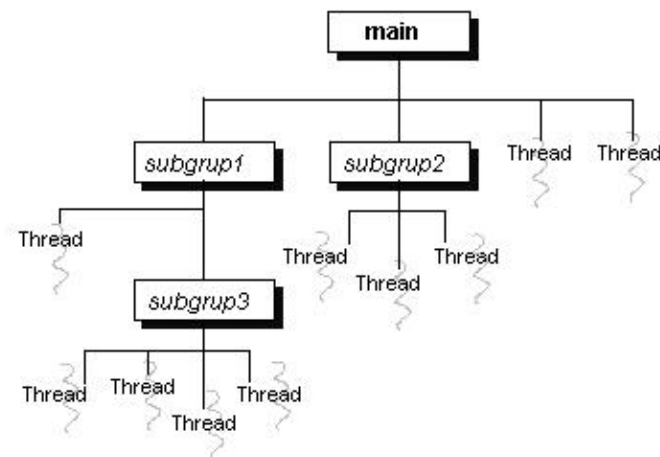
```
public Thread(ThreadGroup group, Runnable target)
public Thread(ThreadGroup group, String name)
public Thread(ThreadGroup group, Runnable target, String name)
```

Fiecare din acești constructori creează un fir de execuție, îl inițializează și îl plasează într-un grup specificat ca argument. Pentru a afla cărui grup aparține un anumit fir de execuție putem folosi metoda `getThreadGroup` a clasei `Thread`. În exemplul următor vor fi create două grupuri, primul cu două fire de execuție iar al doilea cu trei:

```
ThreadGroup grup1 = new ThreadGroup("Producatori");
Thread p1 = new Thread(grup1, "Producator 1");
Thread p2 = new Thread(grup1, "Producator 2");

ThreadGroup grup2 = new ThreadGroup("Consumatori");
Thread c1 = new Thread(grup2, "Consumator 1");
Thread c2 = new Thread(grup2, "Consumator 2");
Thread c3 = new Thread(grup2, "Consumator 3");
```

Un grup poate avea ca părinte un alt grup, ceea ce înseamnă că firele de execuție pot fi plasate într-o ierarhie de grupuri, în care rădăcina este grupul implicit `main`, ca în figura de mai jos:



Să considerăm un exemplu în care listăm firele de execuție active:

Listing 12.10: Folosirea clasei ThreadGroup

```
public class TestThreadGroup {

    static class Dummy implements Runnable {
        public void run() {
            while (true)
                Thread.yield();
        }
    }

    public static void main(String args[]) {

        // Cream o fereastră pentru a fi create
        // automat firele de execuție din AWT
        java.awt.Frame f = new java.awt.Frame("Test");

        // Cream un fir propriu
        new Thread(new Dummy(), "Fir de test").start();

        // Obținem o referință la grupul curent
        Thread firCurent = Thread.currentThread();
        ThreadGroup grupCurent = firCurent.getThreadGroup();

        // Aflăm numărul firelor de execuție active
        int n = grupCurent.activeCount();

        // Enumerăm firele din grup
        Thread[] lista = new Thread[n];
        grupCurent.enumerate(lista);

        // Le afișăm
        for (int i=0; i < n; i++)
            System.out.println("Thread #" + i + "=" +
                lista[i].getName());
    }
}
```

12.5 Comunicarea prin fluxuri de tip ”pipe”

O modalitate deosebit de utilă prin care două fire de execuție pot comunica este realizată prin intermediul *canalelor de comunicații (pipes)*. Acestea sunt implementate prin fluxuri descrise de clasele:

- **PipedReader, PipedWriter** - pentru caractere, respectiv
- **PipedOutputStream, PipedInputStream** - pentru octeți.

Fluxurile ”pipe” de ieșire și cele de intrare pot fi conectate pentru a efectua transmiterea datelor. Acest lucru se realizează uzual prin intermediul constructorilor:

```
public PipedReader(PipedWriterpw)
public PipedWriter(PipedReaderpr)
```

În cazul în care este folosit un constructor fără argumente, conectarea unui flux de intrare cu un flux de ieșire se face prin metoda **connect**:

```
public void connect(PipedWriterpw)
public void connect(PipedReaderpr)
```

Intrucât fluxurile care sunt conectate printr-un pipe trebuie să execute simultan operații de scriere/citire, folosirea lor se va face din cadrul unor fire de execuție.

Funcționarea obiectelor care instanțiază **PipedWriter** și **PipedReader** este asemănătoare cu a canalelor de comunicare UNIX (pipes). Fiecare capăt al unui canal este utilizat dintr-un fir de execuție separat. La un capăt se scriu caractere, la celălalt se citesc. La citire, dacă nu sunt date disponibile firul de execuție se va bloca până ce acestea vor deveni disponibile. Se observă că acesta este un comportament tipic producător-consumator asincron, firele de execuție comunicând printr-un canal.

Realizarea conexiunii se face astfel:

```
PipedWriter pw1 = new PipedWriter();
PipedReader pr1 = new PipedReader(pw1);
// sau
PipedReader pr2 = new PipedReader();
PipedWriter pw2 = new PipedWriter(pr2);
// sau
```

```
PipedReader pr = new PipedReader();
PipedWriter pw = new PipedWriter();
pr.connect(pw) //echivalent cu
pw.connect(pr);
```

Scrierea și citirea pe/de pe canale se realizează prin metodele uzuale `read` și `write`, în toate formele lor.

Să reconsiderăm acum exemplul producător/consumator prezentat anterior, folosind canale de comunicație. Producătorul trimite datele printr-un flux de ieșire de tip `DataOutputStream` către consumator, care le primește printr-un flux de intrare de tip `DataInputStream`. Aceste două fluxuri vor fi interconectate prin intermediul unor fluxuri de tip "pipe".

Listing 12.11: Folosirea fluxurilor de tip "pipe"

```
import java.io.*;

class Producator extends Thread {
    private DataOutputStream out;

    public Producator(DataOutputStream out) {
        this.out = out;
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            try {
                out.writeInt(i);
            } catch (IOException e) {
                e.printStackTrace();
            }
            System.out.println("Producatorul a pus:\t" + i);
            try {
                sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) { }
        }
    }
}

class Consumator extends Thread {
    private DataInputStream in;

    public Consumator(DataInputStream in) {
        this.in = in;
    }
}
```

```
    }

    public void run() {
        int value = 0;
        for (int i = 0; i < 10; i++) {
            try {
                value = in.readInt();
            } catch (IOException e) {
                e.printStackTrace();
            }
            System.out.println("Consumatorul a primit:\t" + value);
        }
    }
}

public class TestPipes {
    public static void main(String[] args)
        throws IOException {

        PipedOutputStream pipeOut = new PipedOutputStream();
        PipedInputStream pipeIn = new PipedInputStream(pipeOut);

        DataOutputStream out = new DataOutputStream(pipeOut);
        DataInputStream in = new DataInputStream(pipeIn);

        Producator p1 = new Producator(out);
        Consumator c1 = new Consumator(in);

        p1.start();
        c1.start();
    }
}
```

12.6 Clasele Timer și TimerTask

Clasa `Timer` oferă o facilitate de a planifica diverse acțiuni pentru a fi realizate la un anumit moment de către un fir de execuție ce rulează în fundal. Acțiunile unui obiect de tip `Timer` sunt implementate ca instanțe ale clasei `TimerTask` și pot fi programate pentru o singură execuție sau pentru execuții repetate la intervale regulate.

Pașii care trebuie făcuți pentru folosirea unui timer sunt:

- Crearea unei subclase `Actiune` a lui `TimerTask` și supreadefinirea metodei `run` ce va conține acțiunea pe care vrem să o planificăm. După cum vom vedea, pot fi folosite și clase anonime.
- Crearea unui fir de execuție prin instanțierea clasei `Timer`;
- Crearea unui obiect de tip `Actiune`;
- Planificarea la execuție a obiectului de tip `Actiune`, folosind metoda `schedule` din clasa `Timer`;

Metodele de planificare pe care le avem la dispoziție au următoarele formate:

```
schedule(TimerTask task, Date time)
schedule(TimerTask task, long delay, long period)
schedule(TimerTask task, Date time, long period)
scheduleAtFixedRate(TimerTask task, long delay, long period)
scheduleAtFixedRate(TimerTask task, Date time, long period)
```

unde, *task* descrie acțiunea ce se va executa, *delay* reprezintă întârzierea față de momentul curent după care va începe execuția, *time* momentul exact la care va începe execuția iar *period* intervalul de timp între două execuții.

După cum se observă, metodele de planificare se împart în două categorii:

- **`schedule`** - planificare cu întârziere fixă: dacă dintr-un anumit motiv acțiunea este întârziată, următoarele acțiuni vor fi și ele întârziate în consecință;
- **`scheduleAtFixedRate`** - planificare cu număr fix de rate: dacă dintr-un anumit motiv acțiunea este întârziată, următoarele acțiuni vor fi executată mai repede, astfel încât numărul total de acțiuni dintr-o perioadă de timp să fie tot timpul același;

Un timer se va opri natural la terminarea metodei sale `run` sau poate fi oprit forțat folosind metoda `cancel`. După oprirea sa el nu va mai putea fi folosit pentru planificarea altor acțiuni. De asemenea, metoda `System.exit` va oprit forțat toate firele de execuție și va termina aplicația curentă.

Listing 12.12: Folosirea claselor Timer și TimerTask

```
import java.util.*;
import java.awt.*;

class Atentie extends TimerTask {
    public void run() {
        Toolkit.getDefaultToolkit().beep();
        System.out.print(".");
    }
}

class Alarma extends TimerTask {
    public String mesaj;
    public Alarma(String mesaj) {
        this.mesaj = mesaj;
    }
    public void run() {
        System.out.println(mesaj);
    }
}

public class TestTimer {
    public static void main(String args[]) {

        // Setam o actiune repetitiva, cu rata fixa
        final Timer t1 = new Timer();
        t1.scheduleAtFixedRate(new Atentie(), 0, 1*1000);

        // Folosim o clasa anonima pentru o alta actiune
        Timer t2 = new Timer();
        t2.schedule(new TimerTask() {
            public void run() {
                System.out.println("S-au scurs 10 secunde.");
                // Oprim primul timer
                t1.cancel();
            }
        }, 10*1000);

        // Setam o actiune pentru ora 22:30
        Calendar calendar = Calendar.getInstance();
        calendar.set(Calendar.HOUR_OF_DAY, 22);
        calendar.set(Calendar.MINUTE, 30);
        calendar.set(Calendar.SECOND, 0);
        Date ora = calendar.getTime();
    }
}
```

```
    Timer t3 = new Timer();  
    t3.schedule(new Alarma("Toti copiii la culcare!"), ora);  
  }  
}
```

Capitolul 13

Programare în rețea

13.1 Introducere

Programarea în rețea implică trimiterea de mesaje și date între aplicații ce rulează pe calculatoare aflate într-o rețea locală sau conectate la Internet. Pachetul care oferă suport pentru scrierea aplicațiilor de rețea este **java.net**. Clasele din acest pachet oferă o modalitate facilă de programare în rețea, fără a fi nevoie de cunoștințe prealabile referitoare la comunicarea efectivă între calculatoare. Cu toate acestea, sunt necesare câteva noțiuni fundamentale referitoare la rețele cum ar fi: protocol, adresa IP, port, socket.

Ce este un protocol ?

Un *protocol* reprezintă o convenție de reprezentare a datelor folosită în comunicarea între două calculatoare. Având în vedere faptul că orice informație care trebuie trimisă prin rețea trebuie serializată astfel încât să poată fi transmisă secvențial, octet cu octet, către destinație, era nevoie de stabilirea unor convenții (procoloale) care să fie folosite atât de calculatorul care trimite datele cât și de cel care le primește, pentru a se ”înțelege” între ele.

Două dintre cele mai utilizate procoloale sunt **TCP** și **UDP**.

- *TCP (Transport Control Protocol)* este un protocol ce furnizează un flux sigur de date între două calculatoare aflate în rețea. Acest protocol asigură stabilirea unei conexiuni permanente între cele două calculatoare pe parcursul comunicației.
- *UDP (User Datagram Protocol)* este un protocol bazat pe pachete inde-

pendente de date, numite *datagrame*, trimise de la un calculator către altul fără a se garanta în vreun fel ajungerea acestora la destinație sau ordinea în care acestea ajung. Acest protocol nu stabilește o conexiună permanentă între cele două calculatoare.

Cum este identificat un calculator în rețea ?

Orice calculator conectat la Internet este identificat în mod unic de adresa sa **IP** (IP este acronimul de la *Internet Protocol*). Aceasta reprezintă un număr reprezentat pe 32 de biți, uzual sub forma a 4 octeți, cum ar fi de exemplu: 193.231.30.131 și este numit adresa IP *numerică*. Corespunzătoare unei adrese numerice există și o adresă IP *simbolică*, cum ar fi `thor.infoiasi.ro` pentru adresa numerică anterioară.

De asemenea, fiecare calculator aflat într-o rețea locală are un nume unic ce poate fi folosit la identificarea locală a acestuia.

Clasa Java care reprezintă noțiunea de adresă IP este **InetAddress**.

Ce este un port ?

Un calculator are în general o singură legătură fizică la rețea. Orice informație destinată unei anumite mașini trebuie deci să specifice obligatoriu adresa IP a acelei mașini. Însă pe un calculator pot exista concurrent mai multe procese care au stabilite conexiuni în rețea, așteptând diverse informații. Prin urmare, datele trimise către o destinație trebuie să specifice pe lângă adresa IP a calculatorului și procesul către care se îndreaptă informațiile respective. Identificarea proceselor se realizează prin intermediul **porturilor**.

Un *port* este un număr pe 16 biți care identifică în mod unic procesele care rulează pe o anumită mașină. Orice aplicație care realizează o conexiune în rețea va trebui să atașeze un număr de port acelei conexiuni. Valorile pe care le poate lua un număr de port sunt cuprinse între 0 și 65535 (deoarece sunt numere reprezentate pe 16 biți), numerele cuprinse între 0 și 1023 fiind însă rezervate unor servicii sistem și, din acest motiv, nu trebuie folosite în aplicații.

Clase de bază din java.net

Clasele din `java.net` permit comunicarea între procese folosind protocoalele

TCP și UDP și sunt prezentate în tabelul de mai jos.

TCP	UDP
URL	DatagramPacket
URLConnection	DatagramSocket
Socket	MulticastSocket
ServerSocket	

13.2 Lucrul cu URL-uri

Termenul *URL* este acronimul pentru *Uniform Resource Locator* și reprezintă o referință (adresă) la o resursă aflată pe Internet. Aceasta este în general un fișier reprezentând o pagină Web, un text, imagine, etc., însă un URL poate referi și interogări la baze de date, rezultate ale unor comenzi executate la distanță, etc. Mai jos, sunt prezentate câteva exemple de URL-uri sunt:

```
http://java.sun.com
http://students.infoiasi.ro/index.html
http://www.infoiasi.ro/~acf/imgs/taz.gif
http://www.infoiasi.ro/~acf/java/curs/9/prog_retea.html#url
```

După cum se observă din exemplele de mai sus, un URL are două componente principale:

- Identificatorul protocolului folosit (http, ftp, etc);
- Numele resursei referite. Acesta are următoarele componente:
 - Numele calculatorului gazdă (www.infoiasi.ro).
 - Calea completă spre resursa referită (acf/java/curs/9/prog_retea.html). Notăția `user` semnifică uzual subdirectorul `html` al directorului rezervat pe un server Web utilizatorului specificat (`HOME`). În cazul în care este specificat doar un director, fișierul ce reprezintă resursa va fi considerat implicit `index.html`.
 - Opțional, o referință de tip *anchor* în cadrul fișierului referit (`#url`).
 - Opțional, portul la care să se realizeze conexiunea.

Clasa care permite lucrul cu URL-uri este **java.net.URL**. Aceasta are mai mulți constructori pentru crearea de obiecte ce reprezintă referințe către resurse aflate în rețea, cel mai uzual fiind cel care primește ca parametru un șir de caractere. În cazul în care șirul nu reprezintă un URL valid va fi aruncată o excepție de tipul `MalformedURLException`.

```
try {
    URL adresa = new URL("http://xyz.abc");
} catch (MalformedURLException e) {
    System.err.println("URL invalid !\n" + e);
}
```

Un obiect de tip URL poate fi folosit pentru:

- Aflarea informațiilor despre resursa referită (numele calculatorului gazdă, numele fișierului, protocolul folosit. etc).
- Citirea printr-un flux a conținutului fișierului respectiv.
- Conectarea la acel URL pentru citirea și scrierea de informații.

Citirea conținutului unui URL

Orice obiect de tip URL poate returna un flux de intrare de tip `InputStream` pentru citirea conținutului său. Secvența standard pentru această operațiune este prezentată în exemplul de mai jos, în care afișăm conținutul resursei specificată la linia de comandă. Dacă nu se specifică nici un argument, va fi afișat fișierul `index.html` de la adresa: `http://www.infoiasi.ro`.

Listing 13.1: Citirea conținutului unui URL

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

public class CitireURL {
    public static void main(String[] args)
        throws IOException{
        String adresa = "http://www.infoiasi.ro";
        if (args.length > 0)
            adresa = args[0];
```

```
BufferedReader br = null;
try {
    URL url = new URL(adresa);
    InputStream in = url.openStream();
    br = new BufferedReader(new InputStreamReader(in));
    String linie;
    while ((linie = br.readLine()) != null) {
        // Afisam linia citita
        System.out.println(linie);
    }
} catch (MalformedURLException e) {
    System.err.println("URL invalid !\n" + e);
} finally {
    br.close();
}
}
```

Conectarea la un URL

Se realizează prin metoda **openConnection** ce stabilește o conexiune bidirecțională cu resursa specificată. Această conexiune este reprezentată de un obiect de tip **URLConnection**, ce permite crearea atât a unui flux de intrare pentru citirea informațiilor de la URL-ul specificat, cât și a unui flux de ieșire pentru scrierea de date către acel URL. Operațiunea de trimitere de date dintr-un program către un URL este similară cu trimiterea de date dintr-un formular de tip FORM aflat într-o pagină HTML. Metoda folosită pentru trimitere este POST.

În cazul trimiterii de date, obiectul URL este uzual un proces ce rulează pe serverul Web referit prin URL-ul respectiv (jsp, servlet, cgi-bin, php, etc).

13.3 Socket-uri

Definiție

Un *socket* (*soclu*) este o abstracțiune software folosită pentru a reprezenta fiecare din cele două "capete" ale unei conexiuni între două procese ce rulează într-o rețea. Fiecare socket este atașat unui port astfel încât să poată identifica unic programul căruia îi sunt destinate datele.

Socket-urile sunt de două tipuri:

- TCP, implementate de clasele `Socket` și `ServerSocket`;
- UDP, implementate de clasa `DatagramSocket`.

O aplicație de rețea ce folosește socket-uri se încadrează în modelul **client/server** de concepere a unei aplicații. În acest model aplicația este formată din două categorii distincte de programe numite *servere*, respectiv *clienți*.

Programele de tip server sunt cele care oferă diverse servicii eventualilor clienți, fiind în stare de așteptare atâta vreme cât nici un client nu le solicită serviciile. Programele de tip client sunt cele care inițiază conversația cu un server, solicitând un anumit serviciu. Uzual, un server trebuie să fie capabil să trateze mai mulți clienți simultan și, din acest motiv, fiecare cerere adresată serverului va fi tratată într-un fir de execuție separat.

Incepând cu versiunea 1.4 a platformei standard Java, există o clasă utilitară care implementează o pereche de tipul (*adresa IP*, *număr port*). Aceasta este **InetSocketAddress** (derivată din `SocketAddress`), obiectele sale fiind utilizate de constructori și metode definite în cadrul claselor ce descriu socket-uri, pentru a specifica cei doi parametri necesari identificării unui proces care trimite sau recepționează date în rețea.

13.4 Comunicarea prin conexiuni

În acest model se stabilește o conexiune TCP între o aplicație client și o aplicație server care furnizează un anumit serviciu. Avantajul protocolul TCP/IP este că asigură realizarea unei comunicări stabile, permanente în rețea, existând siguranța că informațiile trimise de un proces vor fi recepționate corect și complet la destinație sau va fi semnalată o excepție în caz contrar.

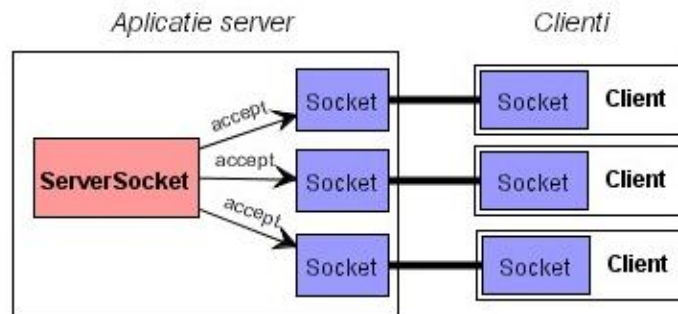
Legătura între un client și un server se realizează prin intermediul a două obiecte de tip **Socket**, câte unul pentru fiecare capăt al "canalului" de comunicație dintre cei doi. La nivelul clientului crearea socketului se realizează specificând adresa IP a serverului și portul la care rulează acesta, constructorul uzual folosit fiind:

```
Socket(InetAddress address, int port)
```

La nivelul serverului, acesta trebuie să creeze întâi un obiect de tip **ServerSocket**. Acest tip de socket nu asigură comunicarea efectivă cu clienții ci este responsabil cu "ascultarea" rețelei și crearea unor obiecte de tip **Socket** pentru fiecare cerere apărută, prin intermediul căruia va fi realizată legătura cu clientul. Crearea unui obiect de tip **ServerSocket** se face specificând portul la care rulează serverul, constructorul folosit fiind:

```
ServerSocket(int port)
```

Metoda clasei **ServerSocket** care așteaptă "ascultă" rețeaua este **accept**. Aceasta blochează procesul părinte până la apariția unui cereri și returnează un nou obiect de tip **Socket** ce va asigura comunicarea cu clientul. Blocarea poate să nu fie permanentă ci doar pentru o anumită perioadă de timp - aceasta va fi specificată prin metoda **setSoTimeout**, cu argumentul dat în milisekunde.



Pentru fiecare din cele două socketuri deschise pot fi create apoi două fluxuri pe octeți pentru citirea, respectiv scrierea datelor. Acest lucru se realizează prin intermediul metodelor **getInputStream**, respectiv **getOutputStream**. Fluxurile obținute vor fi folosite împreună cu fluxuri de procesare care să asigure o comunicare facilă între cele două procese. În funcție de specificul aplicației acestea pot fi perechile:

- **BufferedReader**, **BufferedWriter** și **PrintWriter** - pentru comunicare prin intermediul șirurilor de caractere;
- **DataInputStream**, **DataOutputStream** - pentru comunicare prin date primitive;
- **ObjectInputStream**, **ObjectOutputStream** - pentru comunicare prin intermediul obiectelor;

Structura generală a unui server bazat pe conexiuni este:

```

1. Creeaza un obiect de tip ServerSocket la un anumit port
while (true) {
    2. Asteapta realizarea unei conexiuni cu un client,
       folosind metoda accept;
       (va fi creat un obiect nou de tip Socket)
    3. Trateaza cererea venita de la client:
       3.1 Deschide un flux de intrare si primeste cererea
       3.2 Deschide un flux de iesire si trimite raspunsul
       3.3 Inchide fluxurile si socketul nou creat
}

```

Este recomandat ca tratarea cererilor să se realizeze în fire de execuție separate, pentru ca metoda `accept` să poată fi reapelată cât mai repede în vederea stabilirii conexiunii cu un alt client.

Structura generală a unui client bazat pe conexiuni este:

```

1. Citeste sau declara adresa IP a serverului si portul
   la care acesta ruleaza;
2. Creeaza un obiect de tip Socket cu adresa si portul specificate;
3. Comunica cu serverul:
   3.1 Deschide un flux de iesire si trimite cererea;
   3.2 Deschide un flux de intrare si primeste raspunsul;
   3.3 Inchide fluxurile si socketul creat;

```

În exemplul următor vom implementa o aplicație client-server folosind comunicarea prin conexiuni. Clientul va trimite serverului un nume iar acesta va raspunde prin mesajul "Hello *nume*". Tratarea cererilor se va face în fire de execuție separate.

Listing 13.2: Structura unui server bazat pe conexiuni

```

import java.net.*;
import java.io.*;

class ClientThread extends Thread {
    Socket socket = null;

    public ClientThread(Socket socket) {
        this.socket = socket;
    }
}

```

```

}

public void run() {
    //Executam solicitarea clientului
    String cerere, raspuns;
    try {
        // in este fluxul de intrare de la client
        BufferedReader in = new BufferedReader(new
            InputStreamReader(
                socket.getInputStream() ));

        // out este flux de iesire catre client
        PrintWriter out = new PrintWriter(
            socket.getOutputStream());

        // Primim cerere de la client
        cerere = in.readLine();

        // Trimitem raspuns clientului
        raspuns = "Hello " + cerere + "!";
        out.println(raspuns);
        out.flush();

    } catch (IOException e) {
        System.err.println("Eroare IO \n" + e);
    } finally {
        // Inchidem socketul deschis pentru clientul curent
        try {
            socket.close();
        } catch (IOException e) {
            System.err.println("Socketul nu poate fi inchis \n" +
                e);
        }
    }
}

}

}

public class SimpleServer {

    // Definim portul pe care se gaseste serverul
    // (in afara intervalului 1-1024)
    public static final int PORT = 8100;

    public SimpleServer() throws IOException {
        ServerSocket serverSocket = null;

```



```

try {
    serverSocket = new ServerSocket(PORT);
    while (true) {
        System.out.println("Așteptam un client...");
        Socket socket = serverSocket.accept();

        // Executam solicitarea clientului într-un fir de
        // execuție
        ClientThread t = new ClientThread(socket);
        t.start();
    }
} catch (IOException e) {
    System.err.println("Eroare IO \n" + e);
} finally {
    serverSocket.close();
}
}

public static void main(String[] args) throws IOException {
    SimpleServer server = new SimpleServer();
}
}

```

Listing 13.3: Structura unui client bazat pe conexiuni

```

import java.net.*;
import java.io.*;

public class SimpleClient {

    public static void main(String[] args) throws IOException {
        // Adresa IP a serverului
        String adresaServer = "127.0.0.1";

        // Portul la care serverul ofera serviciul
        int PORT = 8100;

        Socket socket = null;
        PrintWriter out = null;
        BufferedReader in = null;
        String cerere, raspuns;

        try {
            socket = new Socket(adresaServer, PORT);

```

```
    out = new PrintWriter(socket.getOutputStream(), true);
    in = new BufferedReader(new InputStreamReader(
        socket.getInputStream()));

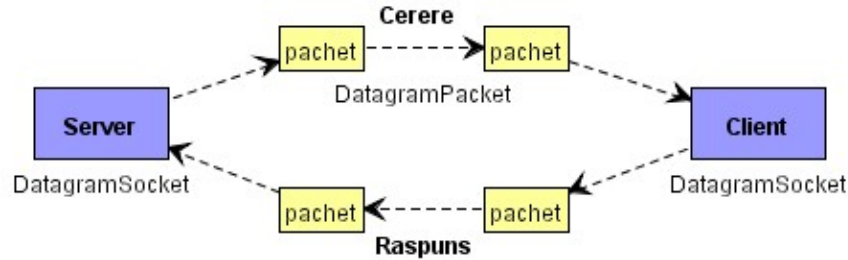
    // Trimitem o cerere la server
    cerere = "Duke";
    out.println(cerere);

    //Asteptam raspunsul de la server ("Hello Duke!")
    raspuns = in.readLine();
    System.out.println(raspuns);

} catch (UnknownHostException e) {
    System.err.println("Serverul nu poate fi gasit \n" + e)
        ;
    System.exit(1);
} finally {
    if (out != null)
        out.close();
    if (in != null)
        in.close();
    if (socket != null)
        socket.close();
}
}
```

13.5 Comunicarea prin datagrame

În acest model nu există o conexiune permanentă între client și server prin intermediul căreia să se realizeze comunicarea. Clientul trimite cererea către server prin intermediul unuia sau mai multor pachete de date independente, serverul le recepționează, extrage informațiile conținute și returnează răspunsul tot prin intermediul pachetelor. Un astfel de pachet se numește *datagramă* și este reprezentat printr-un obiect din clasa **DatagramPacket**. Rutarea datagramelor de la o mașină la alta se face exclusiv pe baza informațiilor conținute de acestea. Primirea și trimiterea datagramelor se realizează prin intermediul unui socket, modelat prin intermediul clasei **DatagramSocket**.



După cum am menționat deja, dezavantajul acestei metode este că nu garantează ajungerea la destinație a pachetelor trimise și nici că vor fi primite în aceeași ordine în care au fost expediate. Pe de altă parte, există situații în care aceste lucruri nu sunt importante și acest model este de preferat celui bazat pe conexiuni care solicită mult mai mult atât serverul cât și clientul. De fapt, protocolul TCP/IP folosește tot pachete pentru trimiterea informațiilor dintr-un nod în altul al rețelei, cu deosebirea că asigură respectarea ordinii de transmitere a mesajelor și verifică ajungerea la destinație a tuturor pachetelor - în cazul în care unul nu a ajuns, acesta va fi retrimis automat.

Clasa `DatagramPacket` conține următorii constructori:

```

DatagramPacket(byte[] buf, int length,
    InetAddress address, int port)
DatagramPacket(byte[] buf, int offset, int length,
    InetAddress address, int port)

DatagramPacket(byte[] buf, int offset, int length,
    SocketAddress address)
DatagramPacket(byte[] buf, int length,
    SocketAddress address)

DatagramPacket(byte[] buf, int length)
DatagramPacket(byte[] buf, int offset, int length)
  
```

Primele două perechi de constructori sunt pentru crearea pachetelor ce vor fi **expediate**, diferența între ele fiind utilizarea claselor `InetAddress`, respectiv `SocketAddress` pentru specificarea adresei destinație.

A trei perechi de constructori este folosită pentru crearea unui pachet în care vor fi **recepționate** date, ei nespecificând vreo sursă sau destinație.

După crearea unui pachet procesul de trimitere și primire a acestuia implică apelul metodelor **send** și **receive** ale clasei **DatagramSocket**. Deoarece toate informații sunt incluse în datagramă, același socket poate fi folosit atât pentru trimiterea de pachete, eventual către destinații diferite, cât și pentru recepționarea acestora de la diverse surse. În cazul în care re folosim pachete, putem schimba conținutul acestora cu metoda **setData**, precum și adresa la care le trimitem prin **setAddress**, **setPort** și **setSocketAddress**.

Extragerea informațiilor conținute de un pachet se realizează prin metoda **getData** din clasa **DatagramPacket**. De asemenea, această clasă oferă metode pentru aflarea adresei IP și a portului procesului care a trimis datagrama, pentru a-i putea răspunde dacă este necesar. Acestea sunt: **getAddress**, **getPort** și **getSocketAddress**.

Listing 13.4: Structura unui server bazat pe datagrama

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

public class DatagramServer {

    public static final int PORT = 8200;
    private DatagramSocket socket = null;
    DatagramPacket cerere, raspuns = null;

    public void start() throws IOException {

        socket = new DatagramSocket(PORT);
        try {
            while (true) {

                // Declaram pachetul in care va fi receptionata
                // cererea
                byte[] buf = new byte[256];
                cerere = new DatagramPacket(buf, buf.length);

                System.out.println("Așteptam un pachet...");
                socket.receive(cerere);

                // Aflăm adresa și portul de la care vine cererea
                InetAddress adresa = cerere.getAddress();
                int port = cerere.getPort();

                // Construim raspunsul
```

```

String mesaj = "Hello " + new String(cerere.getData()
    );
buf = mesaj.getBytes();

// Trimitem un pachet cu raspunsul catre client
raspuns = new DatagramPacket(buf, buf.length, adresa,
    port);
socket.send(raspuns);
}
} finally {
    if (socket != null)
        socket.close();
}
}

public static void main(String[] args) throws IOException {
    new DatagramServer().start();
}
}

```

Listing 13.5: Structura unui client bazat pe datagrame

```

import java.net.*;
import java.io.*;

public class DatagramClient {

    public static void main(String[] args) throws IOException {

        // Adresa IP si portul la care ruleaza serverul
        InetAddress adresa = InetAddress.getByName("127.0.0.1");
        int port=8200;

        DatagramSocket socket = null;
        DatagramPacket packet = null;
        byte buf[];

        try {
            // Construim un socket pentru comunicare
            socket = new DatagramSocket();

            // Construim si trimitem pachetul cu cererea catre
            server
            buf = "Duke".getBytes();

```

```

    packet = new DatagramPacket(buf, buf.length, adresa,
        port);
    socket.send(packet);

    // Asteptam pachetul cu raspunsul de la server
    buf = new byte[256];
    packet = new DatagramPacket(buf, buf.length);
    socket.receive(packet);

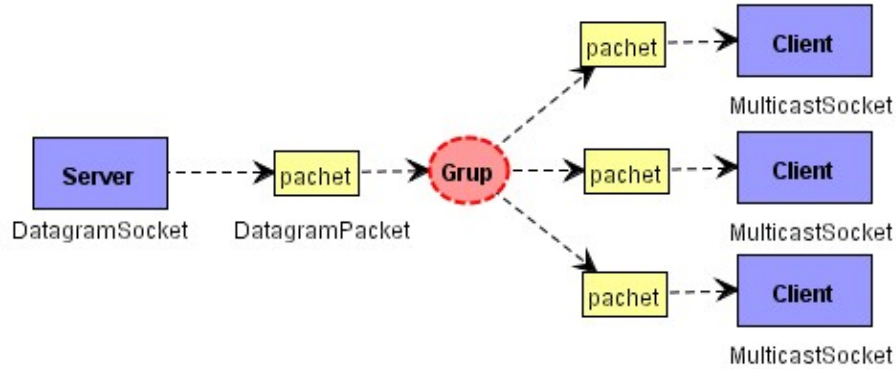
    // Afisam raspunsul ("Hello Duke!")
    System.out.println(new String(packet.getData()));
} finally {
    if (socket != null)
        socket.close();
}
}
}
}

```

13.6 Trimiterea de mesaje către mai mulți clienți

Diverse situații impun gruparea mai multor clienți astfel încât un mesaj (pachet) trimis pe adresa grupului să fie recepționat de fiecare dintre aceștia. Gruparea mai multor programe în vederea trimiterii multiple de mesaje se realizează prin intermediul unui socket special, descris de clasa **MulticastSocket**, extensie a clasei **DatagramSocket**.

Un grup de clienți abonați pentru trimitere multiplă este specificat printr-o adresă IP din intervalul 224.0.0.1 - 239.255.255.255 și un port UDP. Adresa 224.0.0.0 este rezervată și nu trebuie folosită.



Listing 13.6: Inregistrarea unui client într-un grup

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

public class MulticastClient {

    public static void main(String[] args) throws IOException {

        // Adresa IP si portul care reprezinta grupul de clienti
        InetAddress group = InetAddress.getByName("230.0.0.1");
        int port=4444;

        MulticastSocket socket = null;
        byte buf[];

        try {
            // Ne alaturam grupului aflat la adresa si portul
            // specificate
            socket = new MulticastSocket(port);
            socket.joinGroup(group);

            // Asteptam un pachet venit pe adresa grupului
            buf = new byte[256];
            DatagramPacket packet = new DatagramPacket(buf, buf.
                length);

            System.out.println("Asteptam un pachet...");
            socket.receive(packet);
        }
    }
}
```

```

        System.out.println(new String(packet.getData()).trim())
        ;
    } finally {
        if (socket != null) {
            socket.leaveGroup(group);
            socket.close();
        }
    }
}
}
}

```

Listing 13.7: Transmiterea unui mesaj către un grup

```

import java.net.*;
import java.io.*;

public class MulticastSend {

    public static void main(String[] args) throws IOException {

        InetAddress grup = InetAddress.getByName("230.0.0.1");
        int port = 4444;
        byte[] buf;
        DatagramPacket packet = null;

        // Cream un socket cu un numar oarecare
        DatagramSocket socket = new DatagramSocket(0);
        try {
            // Trimitem un pachet catre toti clientii din grup
            buf = (new String("Salut grup!")).getBytes();
            packet = new DatagramPacket(buf, buf.length, grup, port
            );
            socket.send(packet);

        } finally {
            socket.close();
        }
    }
}

```

Capitolul 14

Appleturi

14.1 Introducere

Definiție

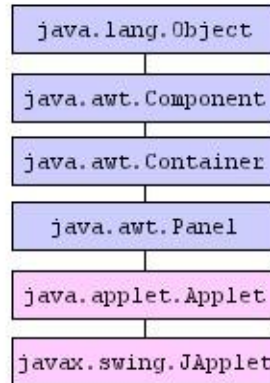
Un *applet* reprezintă un program Java de dimensiuni reduse ce gestionează o suprafață de afișare (container) care poate fi inclusă într-o pagină Web. Un astfel de program se mai numește *miniaplicatie*.

Ca orice altă aplicație Java, codul unui applet poate fi format din una sau mai multe clase. Una dintre acestea este *principală* și extinde clasa **Applet**, aceasta fiind clasa ce trebuie specificată în documentul HTML ce descrie pagina Web în care dorim să includem appletul.

Diferența fundamentală dintre un applet și o aplicație constă în faptul că un applet nu poate fi executat independent, ci va fi executat de browserul în care este încărcată pagina Web ce conține appletul respectiv. O aplicație independentă este executată prin apelul interpretorului *java*, având ca argument numele clasei principale a aplicației, clasa principală fiind cea care conține metoda *main*. Ciclul de viață al unui applet este complet diferit, fiind dictat de evenimentele generate de către browser la vizualizarea documentului HTML ce conține appletul.

Pachetul care oferă suport pentru crearea de appleturi este **java.applet**, cea mai importantă clasă fiind **Applet**. În pachetul **javax.swing** există și clasa **JApplet**, care extinde **Applet**, oferind suport pentru crearea de appleturi pe arhitectura de componente JFC/Swing.

Ierarhia claselor din care derivă appleturile este prezentată în figura de mai jos:



Fiind derivată din clasa `Container`, clasa `Applet` descrie de fapt suprafețe de afișare, asemenea claselor `Frame` sau `Panel`.

14.2 Crearea unui applet simplu

Crearea structurii de fișiere și compilarea applet-urilor sunt identice ca în cazul aplicațiilor. Diferă în schimb structura programului și modul de rulare a acestuia. Să parcurgem în continuare acești pași pentru a realiza un applet extrem de simplu, care afișează o imagine și un șir 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 fișierelor sursă

Ca orice clasă publică, clasa principală a appletului va fi salvată într-un fișier cu același nume și extensia `.java`. Așadar, vom salva clasa de mai sus într-un fișier `FirstApplet.java`.

3. Compilarea

Compilarea se face la fel ca și la aplicațiile independente, folosind compilatorul `javac` apelat pentru fișierul ce conține appletul.

```
javac FirstApplet.java
```

În cazul în care compilarea a reușit va fi generat fișierul `FirstApplet.class`.

4. Rularea appletului

Applet-urile nu rulează 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 operații:

- **Crearea unui fișier HTML** în care vom include applet-ul. Să considerăm fișierul `simply.html`, având conținutul de mai jos:

```
<html>
<head>
  <title>Primul applet Java</title>
</head>
<body>
  <applet code=FirstApplet.class width=400 height=400>
  </applet>
</body>
</html>
```

- **Vizualizarea appletului:** se deschide fisierul `simplu.html` folosind unul din browser-ele amintite sau efectuând apelul:
`appletviewer simplu.html`.

14.3 Ciclul de viață al unui applet

Execuția unui applet începe în momentul în care un browser afișează o pagină Web în care este inclus appletul respectiv și poate trece prin mai multe etape. Fiecare etapă este strâns legată de un eveniment generat de către browser și determină apelarea unei metode specifice din clasa ce implementează appletul.

- **Incărcarea în memorie**
Este creată o instanța a clasei principale a appletului și încarcată în memorie.
- **Inițializarea**
Este apelată metoda `init` ce permite inițializarea diverselor variabile, citirea unor parametri de intrare, etc.
- **Pornirea**
Este apelată metoda `start`
- **Execuția propriu-zisă**
Constă în interacțiunea dintre utilizator și componentele afișate pe suprafața appletului sau în executarea unui anumit cod într-un fir de execuție. În unele situații întreaga execuție a appletului se consumă la etapele de inițializare și pornire.

- **Oprirea temporară**

În cazul în care utilizatorul părăsește pagina Web în care rulează appletul este apelată metoda **stop** a acestuia, dându-i astfel posibilitatea să oprească temporar execuția sa pe perioada în care nu este vizibil, pentru a nu consuma inutil din timpul procesorului. Același lucru se întâmplă dacă fereastra browserului este minimizată. În momentul când pagina Web ce conține appletul devine din nou activă, va fi reapelată metoda **start**.

- **Oprirea definitivă**

La închiderea tuturor instanțelor browserului folosit pentru vizualizare, appletul va fi eliminat din memorie și va fi apelată metoda **destroy** a acestuia, pentru a-i permite să elibereze resursele deținute. Apelul metodei **destroy** este întotdeauna precedat de apelul lui **stop**.

Metodele specifice appleturilor

Așadar, există o serie de metode specifice appleturilor ce sunt apelate automat la diverse evenimente generate de către browser. Acestea sunt definite în clasa **Applet** și sunt enumerate în tabelul de mai jos:

Metoda	Situația în care este apelată
init	La inițializarea appletului. Teoretic, această metodă ar trebui să se apeleze o singură dată, la prima afișare a appletului în pagină, însă, la unele browsere, este posibil ca ea să se apeleze de mai multe ori.
start	Imediat după inițializare și 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) și înainte de destroy .
destroy	La închiderea ultimei instanțe a browserului care a încărcat în memorie clasa principală a appletului.

Atenție

Aceste metode sunt apelate automat de browser și nu trebuie apelate explicit din program !

Structura generală a unui applet

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

public class StructuraApplet extends Applet {

    public void init() {
    }

    public void start() {
    }

    public void stop() {
    }

    public void destroy() {
    }

}
```

14.4 Interfața grafică cu utilizatorul

După cum am văzut, clasa `Applet` este o extensie a superclasei `Container`, ceea ce înseamnă că appleturile sunt, înainte de toate, suprafețe de afișare. Plasarea componentelor, gestionarea poziționării lor și tratarea evenimentelor generate se realizează la fel ca și în cazul aplicațiilor. Uzual, adăugarea componentelor pe suprafața appletului precum și stabilirea obiectelor responsabile cu tratarea evenimentelor generate sunt operațiuni ce vor fi realizate în metoda `init`.

Gestionarul de poziționare implicit este `FlowLayout`, însă acesta poate fi schimbat prin metoda `setLayout`.

Desenarea pe suprafața unui applet

Există o categorie întreagă de appleturi ce nu comunică cu utilizatorul prin intermediul componentelor ci, execuția lor se rezumă la diverse operațiuni 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 suprafața unui applet va fi nevoie să supradefinim această metodă.

```
public void paint(Graphics g) {
    // Desenare
    ...
}
```

În cazul în care este aleasă această soluție, evenimentele tratate uzual vor fi cele generate de mouse sau tastatură.

14.5 Definirea și folosirea parametrilor

Parametrii sunt pentru appleturi ceea ce argumentele de la linia de comandă sunt pentru aplicațiile independente. Ei permit utilizatorului să personalizeze aspectul sau comportarea unui applet fără a-i schimba codul și recompila clasele.

Definirea parametrilor se face în cadrul tagului **APPLET** din documentul HTML ce conține appletul și sunt identificați prin atributul **PARAM**. Fiecare parametru are un nume, specificat prin **NAME** și 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 și în cazul argumentelor trimise aplicațiilor de la linia de comandă, tipul parametrilor este întotdeauna **șir de caractere**, indiferent dacă valoarea este între ghilimele sau nu.

Fiecare applet are și un set de parametri prestabiliți ale căror nume nu vor putea fi folosite pentru definirea de noi parametri folosind metoda de

mai sus. Aceștia apar direct în corpul tagului `APPLET` și definesc informații generale despre applet. Exemple de astfel de parametri sunt `CODE`, `WIDTH` sau `HEIGHT`. Lista lor completă va fi prezentată la descrierea tagului `APPLET`.

Folosirea parametrilor primiți de către un applet se face prin intermediul metodei `getParameter` care primește ca argument numele unui parametru și returnează valoarea acestuia. În 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 dispoziție o "documentație" 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 șiruri. Fiecare element al vectorului este de fapt un vector cu trei elemente de tip `String`, cele trei șiruri reprezentând *numele* parametrului, *tipul* său și o *descriere* a sa. Informațiile furnizate de un applet pot fi citite din browserul folosit pentru vizualizare prin metode specifice acestuia. De exemplu, în `appletviewer` informațiile despre parametri pot fi vizualizate la rubrica *Info* din meniul *Applet*, în Netscape se folosește opțiunea *Page info* din meniul *View*, etc.

Să scriem un applet care să afișeze un text primit ca parametru, folosind un font cu numele și dimensiunea specificate de asemenea ca parametri.

Listing 14.1: Folosirea parametrilor

```
import java.applet.Applet;
import java.awt.*;

public class TestParametri extends Applet {

    String text, numeFont;
    int dimFont;

    public void init() {
        text = getParameter("textAfisat");
        if (text == null)
            text = "Hello"; // valoare implicita

        numeFont = getParameter("numeFont");
        if (numeFont == null)
            numeFont = "Arial";
    }
}
```

```

    try {
        dimFont = Integer.parseInt(getParameter("dimFont"));
    } catch(NumberFormatException e) {
        dimFont = 16;
    }
}

public void paint(Graphics g) {
    g.setFont(new Font(umeFont, Font.BOLD, dimFont));
    g.drawString(text, 20, 20);
}

public String[][] getParameterInfo() {
    String[][] info = {
        // Nume          Tip          Descriere
        {"textAfisat", "String", "Sirul ce va fi afisat"},
        {"umeFont", "String", "Numele fontului"},
        {"dimFont", "int", "Dimensiunea fontului"}
    };
    return info;
}
}

```

14.6 Tag-ul APPLET

Sintaxa completă a tagului APPLET, cu ajutorul căruia pot fi incluse appleturi în cadrul paginilor Web este:

```

<APPLET
    CODE = clasaApplet
    WIDTH = latimeInPixeli
    HEIGHT = inaltimeInPixeli

    [ARCHIVE = arhiva.jar]
    [CODEBASE = URLApplet]
    [ALT = textAlternativ]
    [NAME = numeInstantaApplet]
    [ALIGN = aliniere]
    [VSPACE = spatiuVertical]

```

```

[HSPACE = spatiuOrizontal] >

[< PARAM NAME = parametru1 VALUE = valoare1 >]
[< PARAM NAME = parametru2 VALUE = valoare2 >]
...
[text HTML alternativ]

</APPLET>

```

Atributele puse între paranteze pătrate sunt opționale.

- **CODE** = *clasaApplet*
Numele fișierului ce conține clasa principală a appletului. Acesta va fi căutat în directorul specificat de **CODEBASE**. Nu poate fi absolut și trebuie obligatoriu specificat. Extensia ".class" poate sau nu să apară.
- **WIDTH** = *latimeInPixeli*, **HEIGHT** = *inaltimeInPixeli*
Specifică lățimea și înălțimea suprafeței în care va fi afișat appletul. Sunt obligatorii.
- **ARCHIVE** = *arhiva.jar*
Specifică arhiva în care se găsesc clasele appletului.
- **CODEBASE** = *directorApplet*
Specifică URL-ul la care se găsește clasa appletului. Uzual se exprimă relativ la directorul documentului HTML. În cazul în care lipsește, se consideră implicit URL-ul documentului.
- **ALT** = *textAlternativ*
Specifică textul ce trebuie afișat dacă browserul înțelege tagul **APPLET** dar nu poate rula appleturi Java.
- **NAME** = *numeInstantaApplet*
Oferă posibilitatea de a da un nume respectivei instanțe a appletului, astfel încât mai multe appleturi aflate pe aceeași 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`, semnificațiile lor fiind aceleași ca și la tagul `IMG`.

- **VSPACE** = *spatiuVertical*, **HSPACE** = *spatiuOrizantal*
Specifică numărul de pixeli dintre applet și marginile suprafeței de afișare.
- **PARAM**
Tag-urile `PARAM` sunt folosite pentru specificarea parametrilor unui applet (vezi "Folosirea parametrilor").
- *text HTML alternativ*
Este textul ce va fi afișat în cazul în care browserul nu înțelege tagul `APPLET`. Browserele *Java-enabled* vor ignora acest text.

14.7 Folosirea firelor de execuție în appleturi

La încărcarea unei pagini Web, fiecărui applet îi este creat automat un fir de execuție responsabil cu apelarea metodelor acestuia. Acestea vor rula concurrent după regulile de planificare implementate de mașina virtuală Java a platformei folosite.

Din punctul de vedere al interfeței grafice însă, fiecare applet aflat pe o pagină Web are acces la un **același** fir de execuție, creat de asemenea automat de către browser, și care este responsabil cu desenarea appletului (apelul metodelor `update` și `paint`) precum și cu transmiterea mesajelor generate de către componente. Intrucât toate appleturile de pe pagină "împart" acest fir de execuție, nici unul nu trebuie să îl solicite în mod excesiv, deoarece va provoca funcționarea anormală sau chiar blocarea celorlalte.

În cazul în care dorim să efectuăm operațiuni consumatoare de timp este recomandat să le realizăm într-un alt fir de execuție, pentru a nu bloca interacțiunea utilizatorului cu appletul, redesenarea acestuia sau activitatea celorlalte appleturi de pe pagină.

Să considerăm mai întâi două abordări greșite de lucru cu appleturi. Dorim să creăm un applet care să afișeze la coordonate aleatoare mesajul "Hello", cu pauză de o secundă între două afișări. Prima variantă, greșită de altfel, ar fi:

Listing 14.2: Incorect: blocarea metodei `paint`

```
import java.applet.*;
import java.awt.*;

public class AppletRau1 extends Applet {
    public void paint(Graphics g) {
        while(true) {
            int x = (int)(Math.random() * getWidth());
            int y = (int)(Math.random() * getHeight());
            g.drawString("Hello", x, y);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {}
        }
    }
}
```

Motivul pentru care acest applet nu funcționează corect și probabil va duce la anomalii în funcționarea browserului este că firul de execuție 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 greșită, este următoarea :

Listing 14.3: Incorect: appletul nu termină inițializarea

```
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 soluție corectă problemei propuse, trebuie să folosim un fir de execuție propriu. Structura unui applet care dorește să lanseze un fir de execuție poate avea două forme. În prima situație appletul pornește firul la inițializarea sa iar acesta va rula, indiferent dacă appletul mai este sau nu vizibil, până la oprirea sa naturală (terminarea metodei `run`) sau până la închiderea sesiunii de lucru a browserului.

Listing 14.4: Corect: folosirea unui fir de execuție 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);
    }
}

```

În cazul în care firul de execuție pornit de applet efectuează operații ce

au sens doar dacă appletul este vizibil, cum ar fi animație, ar fi de dorit ca acesta să se oprească atunci când appletul nu mai este vizibil (la apelul metodei `stop`) și să repornească atunci când appletul redevine vizibil (la apelul metodei `start`). Un applet este considerat *activ* imediat după apelul metodei `start` și devine inactiv la apelul metodei `stop`. Pentru a afla dacă un applet este activ se folosește metoda `isActive`.

Să modificăm programul anterior, adăugând și un contor care să numere afișările de mesaje - acesta nu va fi incrementat pe perioada în care appletul nu este activ.

Listing 14.5: Folosirea metodelor `start` și `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);  
  }  
}
```

Atenție

Este posibil ca unele browsere să nu apele metoda `stop` în situațiile prevăzute în specificațiile appleturilor. Din acest motiv, corectitudinea unui applet nu trebuie să se bazeze pe acest mecanism.

14.8 Alte metode oferite de clasa Applet

Pe lângă metodele de bază: `init`, `start`, `stop`, `destroy`, clasa `Applet` oferă metode specifice applet-urilor cum ar fi:

Punerea la dispoziție a unor informații despre applet

Similară cu metoda `getParameterInfo` ce oferă o "documentație" despre parametrii pe care îi acceptă un applet, există metoda `getAppletInfo` ce permite specificarea unor informații legate de applet cum ar fi numele, autorul, versiunea, etc. Metoda returnează un sir de caractere conținând informațiile 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 conține clasa appletului;
- **getDocumentBase** - returnează URL-ul directorului ce conține documentul HTML în care este inclus appletul respectiv.

Aceste metode sunt foarte utile deoarece permit specificarea relativă a unor fișiere folosite de un applet, cum ar fi imagini sau sunete.

Afișarea unor mesaje în bara de stare a browserului

Acest lucru se realizează cu metoda **showStatus**

```
public void init() {  
    showStatus("Initializare applet...");  
}
```

Afișarea imaginilor

Afișarea imaginilor într-un applet se face fie prin intermediul unei componente ce permite acest lucru, cum ar fi o suprafață de desenare de tip **Canvas**, fie direct în metoda **paint** a applet-ului, folosind metoda **drawImage** a clasei **Graphics**. În ambele cazuri, obținerea unei referințe la imaginea respectivă se va face cu ajutorul metodei **getImage** din clasa **Applet**. Aceasta poate primi ca argument fie adresa URL absolută a fișierului ce reprezintă imaginea, fie calea relativă la o anumită adresă URL, cum ar fi cea a directorului în care se găsește documentul HTML ce conține appletul (**getDocumentBase**) sau a directorului în care se găsește clasa appletului (**getCodeBase**).

Listing 14.6: Afișarea 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 execuție

Contextul de execuție al unui applet se referă la pagina în care acesta rulează, eventual împreună cu alte appleturi, și este descris de interfața **AppletContext**. Crearea unui obiect ce implementează această interfață se realizează de către browser, la apelul metodei **getAppletContext** a clasei **Applet**. Prin intermediul acestei interfețe un applet poate "vedea" în jurul sau, putând comunica cu alte applet-uri aflate pe aceeași pagină sau cere browser-ului să deschidă diverse documente.

```
AppletContext contex = getAppletContext();
```

Afișarea unor documente în browser

Se face cu metoda **showDocument** ce primește adresa URL a fișierului ce conține documentul pe care dorim sa-l deschidem (text, html, imagine, etc). Această metodă este accesată prin intermediul contextului de execuție 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 aceeași pagină și apelarea unei metode sau setarea unei variabile publice a acestuia. Identificarea se face prin intermediu numelui pe care orice instanța a unui applet îl poate specifica prin atributul **NAME**.

Obținerea unei referințe la un applet al cărui nume îl cunoaștem sau obținerea unei enumerări a tuturor applet-urilor din pagină se fac prin intermediul contextului de execuție, folosind metodele **getApplet**, respectiv **getApplets**.

Redarea sunetelor

Clasa **Applet** oferă și posibilitatea redării de sunete în format **.au**. Acestea sunt descrise prin intermediul unor obiecte ce implementează interfața **AudioClip** din pachetul **java.applet**. Pentru a reda un sunet aflat într-un fișier ".au" la un anumit URL există două posibilități:

- Folosirea metodei **play** din clasa **Applet** care primește 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** și **stop** pentru acesta.

Listing 14.7: Redarea sunetelor

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

public class Sunete extends Applet implements ActionListener{
    Button play = new Button("Play");
    Button loop = new Button("Loop");
    Button stop = new Button("Stop");
    AudioClip clip = null;

    public void init() {
        // Fisierul cu sunetul trebuie sa fie in acelasi
        // director cu appletul
        clip = getAudioClip(getCodeBase(), "sunet.au");
        add(play);
        add(loop);
        add(stop);

        play.addActionListener(this);
        loop.addActionListener(this);
        stop.addActionListener(this);
    }
}
```

```
}  
  
public void actionPerformed(ActionEvent e) {  
    Object src = e.getSource();  
    if (src == play)  
        clip.play();  
    else if (src == loop)  
        clip.loop();  
    else if (src == stop)  
        clip.stop();  
}  
}
```

În cazul în care appletul folosește mai multe tipuri de sunete, este recomandat ca încărcarea acestora să fie făcută într-un fir de execuție separat, pentru a nu bloca temporar activitatea firească a programului.

14.9 Arhivarea appleturilor

După cum am văzut, pentru ca un applet aflat pe o pagină Web să poată fi executat codul său va fi transferat de pe serverul care găzduiește pagina Web solicitată pe mașina clientului. Deoarece transferul datelor prin rețea este un proces lent, cu cât dimensiunea fișierelor care formează appletul este mai redusă, cu atât încărcarea acestuia se va face mai repede. Mai mult, dacă appletul conține și alte clase în afară de cea principală sau diverse resurse (imagini, sunete, etc), acestea vor fi transferate prin rețea 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 fișierele necesare acestuia.

Arhivarea fișierelor unui applet se face cu utilitarul **jar**, oferit în distribuția 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 și numele arhivei care o conține:

```
<applet archive=arhiva.jar code=ClasaPrincipala  
width=400 height=200 />
```

14.10 Restricții de securitate

Deoarece un applet se execută pe mașina utilizatorului care a solicitat pagina Web ce conține appletul respectiv, este foarte important să existe anumite restricții de securitate care să controleze activitatea acestuia, pentru a preveni acțiuni rău intenționate, cum ar fi ștergeri de fișiere, 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 excepții de tip **Security Exception** în cazul în care una din acestea încearcă să efectueze o operație nepermisă.

Un applet nu poate să:

- Citească sau să scrie fișiere pe calculatorul pe care a fost încărcat (client).
- Deschidă conexiuni cu alte mașini în afară de cea de pe care provine (host).
- Pornească programe pe mașina client.
- Citească diverse proprietăți ale sistemului de operare al clientului.

Ferestrele folosite de un applet, altele decât cea a browserului, vor arăta altfel decât într-o aplicație obișnuită, indicând faptul că au fost create de un applet.

14.11 Appleturi care sunt și aplicații

Deoarece clasa **Applet** este derivată din **Container**, deci și din **Component**, ea descrie o suprafață de afișare care poate fi inclusă ca orice altă componentă într-un alt container, cum ar fi o fereastră. Un applet poate funcționa și ca o aplicație independentă astfel:

- Adăugăm metoda `main` clasei care descrie appletul, în care vom face operațiunile următoare.
- Creăm o instanță a appletului și o adăugăm pe suprafața unei ferestre.
- Apelăm metodele `init` și `start`, care ar fi fost apelate automat de către browser.
- Facem fereastra vizibilă.

Listing 14.8: Applet și aplicație

```
import java.applet.Applet;
import java.awt.*;

public class AppletAplicatie extends Applet {

    public void init() {
        add(new Label("Applet si aplicatie"));
    }

    public static void main(String args[]) {
        AppletAplicatie applet = new AppletAplicatie();
        Frame f = new Frame("Applet si aplicatie");
        f.setSize(200, 200);

        f.add(applet, BorderLayout.CENTER);
        applet.init();
        applet.start();

        f.show();
    }
}
```

Capitolul 15

Lucrul cu baze de date

15.1 Introducere

15.1.1 Generalități despre baze de date

Aplicațiile care folosesc baze de date sunt, în general, aplicații complexe folosite pentru gestionarea unor informații de dimensiuni mari într-o manieră sigură și eficientă.

Ce este o bază de date ?

La nivelul cel mai general, o *bază de date* reprezintă o modalitate de stocare a unor informații (date) pe un suport extern, cu posibilitatea regăsirii acestora. Uzual, o bază de date este memorată într-unul sau mai multe fișiere.

Modelul clasic de baze de date este cel *relațional*, în care datele sunt memorate în tabele. Un tabel reprezintă o structură de date formată dintr-o mulțime de articole, fiecare articol având definite o serie de attribute - aceste attribute corespund coloanelor tabelului, în timp ce o linie va reprezenta un articol. Pe lângă tabele, o bază de date mai poate conține: proceduri și funcții, utilizatori și grupuri de utilizatori, tipuri de date, obiecte, etc.

Dintre producătorii cei mai importanți de baze de date amintim companiile Oracle, Sybase, IBM, Informix, Microsoft, etc. fiecare furnizând o serie întreagă de produse și utilitare pentru lucrul cu baze de date. Aceste produse sunt în general referite prin termenii *DBMS* (Database Management System) sau, în traducere, *SGBD* (Sistem de Gestiune a Bazelor de Date). În acest capitol vom analiza lucrul cu baze de date din perspectiva programării

în limbajul Java, fără a descrie particularități ale unei soluții de stocare a datelor anume. Vom vedea că, folosind Java, putem crea aplicații care să ruleze fără nici o modificare folosind diverse tipuri de baze care au aceeași structură, ducând în felul acesta noțiunea de portabilitate și mai departe.

Crearea unei baze de date

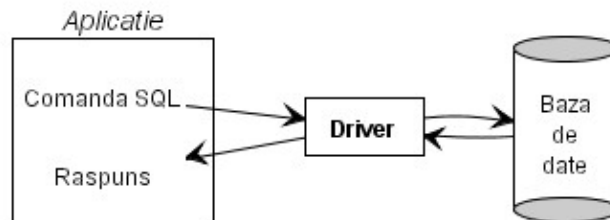
Crearea unei baze de date se face uzual folosind aplicații specializate oferite de producătorul tipului respectiv de sistem de gestiune a datelor, dar există și posibilitatea de a crea o baza folosind un script SQL. Acest aspect ne va preocupa însă mai puțin, exemplele prezentate presupunând că baza a fost creată deja și are o anumită structură specificată.

Accesul la baza de date

Se face prin intermediul unui **driver** specific tipului respectiv de SGBD. Acesta este responsabil cu accesul efectiv la datele stocate, fiind legatura dintre aplicație și baza de date.

Limbajul SQL

SQL (Structured Query Language) reprezintă un limbaj de programare ce permite interogarea și actualizarea informațiilor din baze de date relaționale. Acesta este standardizat astfel încât diverse tipuri de drivere să se comporte identic, oferind astfel o modalitate unitară de lucru cu baze de date.



15.1.2 JDBC

JDBC (Java Database Connectivity) este o interfață standard SQL de acces la baze de date. JDBC este constituită dintr-un set de clase și interfețe

scrise în Java, furnizând mecanisme standard pentru proiectanții aplicațiilor ce folosesc de baze de date.

Folosind JDBC este ușor să transmitem secvențe SQL către baze de date relaționale. Cu alte cuvinte, nu este necesar să scriem un program pentru a accesa o bază de date Oracle, alt program pentru a accesa o bază de date Sybase și așa mai departe. Este de ajuns să scriem un singur program folosind API-ul JDBC și acesta va fi capabil să comunice cu drivere diferite, trimițând secvențe SQL către baza de date dorită. Bineînțeles, scriind codul sursă în Java, ne este asigurată portabilitatea programului. Deci, iată două motive puternice care fac combinația Java - JDBC demnă de luat în seamă.

Pachetele care oferă suport pentru lucrul cu baze de date sunt **java.sql** ce reprezintă nucleul tehnologiei JDBC și, preluat de pe platforma J2EE, **javax.sql**.

În linii mari, API-ul JDBC oferă următoarele facilități:

1. Stabilirea unei conexiuni cu o bază de date.
2. Efectuarea de secvențe SQL.
3. Prelucrarea rezultatelor obținute.

15.2 Conectarea la o bază de date

Procesul de conectare la o bază de date implică efectuarea a două operații:

1. Înregistrarea unui driver corespunzător.
2. Realizarea unei conexiuni propriu-zise.

Definiție

O *conexiune (sesiune)* la o bază de date reprezintă un context prin care sunt trimise secvențe SQL și primite rezultate. Într-o aplicație pot exista simultan mai multe conexiuni la baze de date diferite sau la aceeași bază.

Clasele și interfețele responsabile cu realizarea unei conexiuni sunt:

- **DriverManager** - este clasa ce se ocupă cu înregistrarea driverelor ce vor fi folosite în aplicație;

- **Driver** - interfața pe care trebuie să o implementeze orice clasă ce descrie un driver;
- **DriverPropertyInfo** - prin intermediul acestei clase pot fi specificate diverse proprietăți ce vor fi folosite la realizarea conexiunilor;
- **Connection** - descrie obiectele ce modelează o conexiune propriu-zisă cu baza de date.

15.2.1 Inregistrarea unui driver

Primul lucru pe care trebuie să-l facă o aplicație în procesul de conectare la o bază de date este să înregistreze la mașina virtuală ce rulează aplicația driverul JDBC responsabil cu comunicarea cu respectiva bază de date. Acest lucru presupune încărcarea în memorie a clasei ce implementează driver-ul și poate fi realizată în mai multe modalități.

- Folosirea clasei **DriverManager**:
`DriverManager.registerDriver(new TipDriver());`
- Folosirea metodei **Class.forName** ce apelează **ClassLoader**-ul mașinii virtuale:
`Class.forName("TipDriver");`
`Class.forName("TipDriver").newInstance();`
- Setarea proprietății sistem **jdbc.drivers**, care poate fi realizată în două feluri:
 - De la linia de comandă:
`java -Djdbc.drivers=TipDriver Aplicatie`
 - Din program:
`System.setProperty("jdbc.drivers", "TipDriver");`

Folosind această metodă, specificarea mai multor drivere se face separând numele claselor cu punct și virgulă.

Dacă sunt înregistrate mai multe drivere, ordinea de precedență în alegerea driverului folosit la crearea unei noi conexiuni este:

- 1) Driverele înregistrate folosind proprietatea `jdbc.drivers` la inițializarea mașinii virtuale ce va rula procesul.
- 2) Driverele înregistrate dinamic din aplicație.

15.2.2 Specificarea unei baze de date

O dată ce un driver JDBC a fost înregistrat, acesta poate fi folosit la stabilirea unei conexiuni cu o bază de date. Având în vedere faptul ca pot exista mai multe drivere încărcate în memorie, trebuie să avem posibilitatea de a specifica pe lângă un identificator al bazei de date și driverul ce trebuie folosit. Aceasta se realizează prin intermediul unei adrese specifice, numită **JDBC URL**, ce are următorul format:

jdbc:sub-protocol:identificator

Câmpul *sub-protocol* denumește tipul de driver ce trebuie folosit pentru realizarea conexiunii și poate fi *odbc*, *oracle*, *sybase*, *db2* și așa mai departe.

Identificatorul bazei de date este un indicator specific fiecărui driver corespunzător bazei de date cu care aplicația dorește să interacționeze. În funcție de tipul driver-ului acest identificator poate include numele unei mașini gazdă, un număr de port, numele unui fișier sau al unui director, etc., ca în exemplele de mai jos:

```
jdbc:odbc:test
jdbc:oracle:thin@persistentjava.com:1521:test
jdbc:sybase:test
jdbc:db2:test
```

Subprotocolul *odbc* este un caz special, în sensul că permite specificarea în cadrul URL-ului a unor atribute ce vor fi realizate la crearea unei conexiuni. Sintaxa completa subprotocolului *odbc* este:

jdbc:odbc:identificator[;atribut=valoare]*

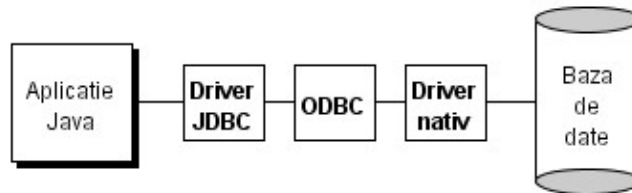
```
jdbc:odbc:test
jdbc:odbc:test;CacheSize=20;ExtensionCase=LOWER
jdbc:odbc:test;UID=duke;PWD=java
```

La primirea unui JDBC URL, *DriverManager*-ul va parcurge lista driverelor înregistrate în memorie, pâna când unul dintre ele va recunoaște URL-ul respectiv. Dacă nu exista nici unul potrivit, atunci va fi lansată o excepție de tipul *SQLException*, cu mesajul "no suitable driver".

15.2.3 Tipuri de drivere

Tipurile de drivere existente ce pot fi folosite pentru realizarea unei conexiuni prin intermediul JDBC se împart în următoarele categorii:

Tip 1. *JDBC-ODBC Bridge*



Acest tip de driver permite conectarea la o bază de date care a fost înregistrată în prealabil în ODBC. ODBC (Open Database Conectivity) reprezintă o modalitate de a uniformiza accesul la baze de date, asociind acestora un identificator DSN (Data Source Name) și diverși parametri necesari conectării. Conectarea efectivă la baza de date se va face prin intermediul acestui identificator, driver-ul ODBC efectuând comunicarea cu driverul nativ al bazei de date.

Deși simplu de utilizat, soluția JDBC-ODBC nu este portabilă și comunicarea cu baza de date suferă la nivelul vitezei de execuție datorită multiplelor redirectări între drivere. De asemenea, atât ODBC-ul cât și driver-ul nativ trebuie să existe pe mașina pe care rulează aplicația.

Clasa Java care descrie acest tip de driver JDBC este:

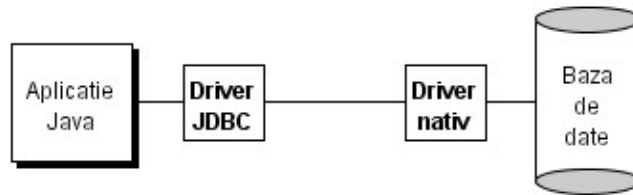
`sun.jdbc.odbc.JdbcOdbcDriver`

și este inclusă în distribuția standard J2SDK. Specificarea bazei de date se face printr-un URL de forma:

`jdbc:odbc:identificator`

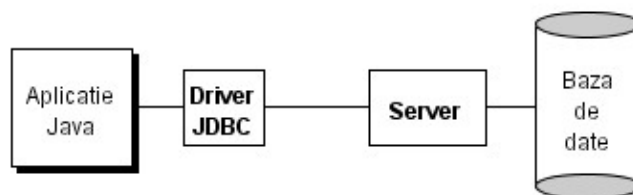
unde *identificator* este profilul (DSN) creat bazei de date în ODBC.

Tip 2. *Driver JDBC - Driver nativ*



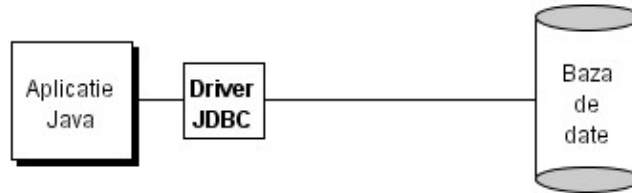
Acest tip de driver transformă cererile JDBC direct în apeluri către driverul nativ al bazei de date, care trebuie instalat în prealabil. Clase Java care implementează astfel de drivere pot fi procurate de la producătorii de SGBD-uri, distribuția standard J2SDK neincluzând nici unul.

Tip 3. *Driver JDBC - Server*



Acest tip de driver transformă cererile JDBC folosind un protocol de rețea independent, acestea fiind apoi transformate folosind o aplicație server într-un protocol specific bazei de date. Introducerea serverului ca nivel intermediar aduce flexibilitate maximă în sensul că vor putea fi realizate conexiuni cu diferite tipuri de baze, fără nici o modificare la nivelul clientului. Protocolul folosit este specific fiecărui producător.

Tip 4. *Driver JDBC nativ*



Acest tip de driver transformă cererile JDBC direct în cereri către baza de date folosind protocolul de rețea al acesteia. Această soluție este cea mai rapidă, fiind preferată la dezvoltarea aplicațiilor care manevrează volume mari de date și viteza de execuție este critică. Driverere de acest tip pot fi procurate de la diverși producători de SGBD-uri.

15.2.4 Realizarea unei conexiuni

Metoda folosită pentru realizarea unei conexiuni este **getConnection** din clasa **DriverManager** și poate avea mai multe forme:

```

Connection c = DriverManager.getConnection(url);
Connection c = DriverManager.getConnection(url, username, password);
Connection c = DriverManager.getConnection(url, dbproperties);
  
```

Stabilirea unei conexiuni folosind driverul JDBC-ODBC

```

String url = "jdbc:odbc:test" ;
// sau url = "jdbc:odbc:test;UID=duke;PWD=java" ;
try {
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
} catch(ClassNotFoundException e) {
    System.err.println("ClassNotFoundException: " + e) ;
    return ;
}

Connection con ;
try {
    con = DriverManager.getConnection(url, "duke", "java");
} catch(SQLException e) {
    System.err.println("SQLException: " + e);
  
```

```

} finally {
    try{
        con.close ;
    } catch(SQLException e) {
        System.err.println(SQLException: " + e) ;
    }
}

```

Stabilirea unei conexiuni folosind un driver MySql

Folosirea diferitelor tipuri de drivere implică doar schimbarea numelui clasei ce reprezintă driverul și a modalității de specificare a bazei de date.

```

String url = "jdbc:mysql://localhost/test" ;
// sau url = "jdbc:mysql://localhost/test?user=duke&password=java";
try {
    Class.forName("com.mysql.jdbc.Driver") ;
} catch(ClassNotFoundException e) {
    ...

```

O conexiune va fi folosită pentru:

- Crearea de secvențe SQL utilizate pentru interogarea sau actualizarea bazei.
- Aflarea unor informații legate de baza de date (meta-date).

De asemenea, clasa **Connection** asigură facilități pentru controlul tranzacțiilor din memorie către baza de date prin metodele **commit**, **rollback**, **setAutoCommit**.

Inchiderea unei conexiuni se realizează prin metoda **close**.

15.3 Efectuarea de secvențe SQL

O dată făcută conectarea cu metoda **DriverManager.getConnection**, se poate folosi obiectul **Connection** rezultat pentru a se crea obiecte de tip **Statement**, **PreparedStatement** sau **CallableStatement** cu ajutorul cărora putem trimite secvențe SQL către baza de date. Cele mai uzuale comenzi SQL sunt cele folosite pentru:

- Interogarea bazei de date: **SELECT**

- Actualizarea datelor: `INSERT`, `UPDATE`, `DELETE`
- Actualizarea structurii: `CREATE`, `ALTER`, `DROP` - acestea mai sunt numite instrucțiuni DDL (Data Definition Language)
- Apelarea unei proceduri stocate: `CALL`

După cum vom vedea, obținerea și prelucrarea rezultatelor unei interogări este realizată prin intermediul obiectelor de tip **ResultSet**.

15.3.1 Interfața Statement

Interfața **Statement** oferă metodele de bază pentru trimiterea de secvențe SQL către baza de date și obținerea rezultatelor, celelalte două interfețe: **PreparedStatement** și **CallableStatement** fiind derivate din aceasta.

Crearea unui obiect **Statement** se realizează prin intermediul metodei **createStatement** a clasei **Connection**, fără nici un argument:

```
Connection con = DriverManager.getConnection(url);  
Statement stmt = con.createStatement();
```

Execuția unei secvențe SQL poate fi realizată prin intermediul a trei metode:

1. executeQuery

Este folosită pentru realizarea de interogări de tip **SELECT**. Metoda returnează un obiect de tip **ResultSet** ce va conține sub o formă tabelară rezultatul interogării.

```
String sql = "SELECT * FROM persoane";  
ResultSet rs = stmt.executeQuery(sql);
```

2. executeUpdate

Este folosită pentru actualizarea datelor (**INSERT**, **UPDATE**, **DELETE**) sau a structurii bazei de date (**CREATE**, **ALTER**, **DROP**). Metoda va returna un întreg ce semnifică numărul de linii afectate de operațiunea de actualizare a datelor, sau 0 în cazul unei instrucțiuni DDL.

```
String sql = "DELETE FROM persoane WHERE cod > 100";
int linii = stmt.executeUpdate(sql);
// Nr de articole care au fost afectate (sterse)

sql = "DROP TABLE temp";
stmt.executeUpdate(sql); // returneaza 0
```

3. execute

Această metodă va fi folosită doar dacă este posibil ca rezultatul unei interogări să fie format din două sau mai multe obiecte de tip `ResultSet` sau rezultatul unei actualizări să fie format din mai multe valori, sau o combinație între aceste cazuri. Această situație, deși mai rară, este posibilă atunci când sunt executate proceduri stocate sau secvențe SQL cunoscute abia la momentul execuției, programatorul neștiind deci dacă va fi vorba de o actualizare a datelor sau a structurii. Metoda întoarce `true` dacă rezultatul obținut este format din obiecte de tip `ResultSet` și `false` dacă e format din întregi. În funcție de aceasta, pot fi apelate metodele: `getResultSet` sau `getUpdateCount` pentru a afla efectiv rezultatul comenzii SQL. Pentru a prelua toate rezultatele va fi apelată metoda `getMoreResults`, după care vor fi apelate din nou metodele amintite, până la obținerea valorii `null`, respectiv `-1`. Secvența completă de tratare a metodei `execute` este prezentată mai jos:

```
String sql = "comanda SQL necunoscuta";
stmt.execute(sql);
while(true) {
    int rowCount = stmt.getUpdateCount();
    if(rowCount > 0) {
        // Este o actualizare datelor
        System.out.println("Linii afectate = " + rowCount);
        stmt.getMoreResults();
        continue;
    }

    if(rowCount = 0) {
        // Comanda DDL sau nici o linie afectata
        System.out.println("Comanda DDL sau 0 actualizari");
    }
}
```

```

        stmt.getMoreResults();
        continue;
    }

    // rowCount este -1
    // Avem unul sau mai multe ResultSet-uri
    ResultSet rs = stmt.getResultSet();
    if(rs != null) {
        // Proceaseaza rezultatul
        ...
        stmt.getMoreResults();
        continue;
    }

    // Nu mai avem nici un rezultat
    break;
}

```

Folosind clasa `Statement`, în cazul în care dorim să introducem valorile unor variabile într-o secvență SQL, nu avem altă soluție decât să creăm un șir de caractere compus din instrucțiuni SQL și valorile variabilelor:

```

int cod = 100;
String nume = "Popescu";
String sql = "SELECT * FROM persoane WHERE cod=" + cod +
    " OR nume='" + nume + "'";
ResultSet rs = stmt.executeQuery(sql);

```

15.3.2 Interfața `PreparedStatement`

Interfața `PreparedStatement` este derivată din `Statement`, fiind diferită de aceasta în următoarele privințe:

- Instanțele de tip `PreparedStatement` conțin secvențe SQL care au fost deja compilate (sunt "pregătite").
- O secvență SQL specificată unui obiect `PreparedStatement` poate să aibă unul sau mai mulți parametri de intrare, care vor fi specificați prin intermediul unui semn de întrebare ("?") în locul fiecăruia dintre

ei. Înainte ca secvența SQL să poată fi executată fiecărui parametru de intrare trebuie să i se atribuie o valoare, folosind metode specifice acestei clase.

Execuția repetată a aceleiași secvențe SQL, dar cu parametri diferiți, va fi în general mai rapidă dacă folosim `PreparedStatement`, deoarece nu mai trebuie să creăm câte un obiect de tip `Statement` pentru fiecare apel SQL, ci refolosim o singură instanță precompilată furnizându-i doar alte argumente.

Crearea unui obiect de tip `PreparedStatement` se realizează prin intermediul metodei `prepareStatement` a clasei `Connection`, specificând ca argument o secvență SQL ce conține câte un semn de întrebare pentru fiecare parametru de intrare:

```
Connection con = DriverManager.getConnection(url);
String sql = "UPDATE persoane SET nume=? WHERE cod=?";
Statement pstmt = con.prepareStatement(sql);
```

Obiectul `pstmt` conține o comandă SQL precompilată care este trimisă imediat către baza de date, unde va aștepta parametri de intrare pentru a putea fi executată.

Trimiterea parametrilor se realizează prin metode de tip `setXXX`, unde `XXX` este tipul corespunzător parametrului, iar argumentele metodei sunt *numărul de ordine* al parametrului de intrare (al semnului de întrebare) și *valoarea* pe care dorim să o atribuim.

```
pstmt.setString(1, "Ionescu");
pstmt.setInt(2, 100);
```

După stabilirea parametrilor de intrare secvența SQL poate fi executată. Putem apoi stabili alte valori de intrare și refolosi obiectul `PreparedStatement` pentru execuții repetate ale comenzii SQL. Este însă posibil ca SGBD-ul folosit să nu suporte acest tip de operațiune și să nu rețină obiectul precompilat pentru execuții ulterioare. În această situație folosirea interfeței `PreparedStatement` în loc de `Statement` nu va îmbunătăți în nici un fel performanța codului, din punctul de vedere al vitezei de execuție a acestuia.

Execuția unei secvențe SQL folosind un obiect `PreparedStatement` se realizează printr-una din metodele `executeQuery`, `executeUpdate` sau `execute`, semnificațiile lor fiind aceleași ca și în cazul obiectelor de tip `Statement`, cu singura deosebire că în cazul de față ele nu au nici un argument.

```
String sql = "UPDATE persoane SET nume=? WHERE cod=?";
Statement pstmt = con.prepareStatement(sql);
pstmt.setString(1, "Ionescu");
pstmt.setInt(2, 100);
pstmt.executeUpdate();

pstmt.setString(1, "Popescu");
pstmt.setInt(2, 200);
pstmt.executeUpdate();

sql = "SELECT * from persoane WHERE cod >= ?";
pstmt = con.prepareStatement(sql);
pstmt.setInt(1, 100);
ResultSet rs = pstmt.executeQuery();
```

Fiecărui tip Java îi corespunde un tip generic SQL. Este responsabilitatea programatorului să se asigure că folosește metoda adecvată de tip `setXXX` la stabilirea valorii unui parametru de intrare. Lista tuturor tipurilor generice disponibile, numite și *tipuri JDBC*, este definită de clasa **Types**, prin constantelor declarate de aceasta. Metoda **setObject** permite specificarea unor valori pentru parametrii de intrare, atunci când dorim să folosim maparea implicită între tipurile Java și cele JDBC sau atunci când dorim să precizăm explicit un tip JDBC.

```
pstmt.setObject(1, "Ionescu", Types.CHAR);
pstmt.setObject(2, 100, Types.INTEGER); // sau doar
pstmt.setObject(2, 100);
```

Folosind metoda **setNull** putem să atribuim unui parametru de intrare valoare SQL NULL, trebuind însă să specificăm și tipul de date al coloanei în care vom scrie această valoare. Același lucru poate fi realizat cu metode de tipul `setXXX` dacă argumentul folosit are valoarea `null`.

```
pstmt.setNull(1, Types.CHAR);
pstmt.setInt(2, null);
```

Cu ajutorul metodelor **setBytes** sau **setString** avem posibilitatea de a specifica date de orice dimensiuni ca valori pentru anumite articole din baza de date. Există însă situații când este de preferat ca datele de mari dimensiuni să fie transferate pe "bucăți" de o anumită dimensiune. Pentru a realiza

acest lucru API-ul JDBC pune la dispoziție metodele **setBinaryStream**, **setAsciiStream** și **setUnicodeStream** care atașează un flux de intrare pe octeți, caractere ASCII, respectiv UNICODE, unui parametru de intrare. Pe măsură ce sunt citite date de pe flux, ele vor fi atribuite parametrului. Exemplul de mai jos ilustrează acest lucru, atribuind coloanei *continut* conținutul unui anumit fișier:

```
File file = new File("date.txt");
int fileLength = file.length();
InputStream fin = new FileInputStream(file);
java.sql.PreparedStatement pstmt = con.prepareStatement(
    "UPDATE fisiere SET continut = ? WHERE nume = 'date.txt'");
pstmt.setUnicodeStream (1, fin, fileLength);
pstmt.executeUpdate();
```

La execuția secvenței, fluxul de intrare va fi apelat repetat pentru a furniza datele ce vor fi scrise în coloana *continut* a articolului specificat. Observați că este necesar înă să știm dinainte dimensiunea datelor ce vor fi scrise, acest lucru fiind solicitat de unele tipuri de baze de date.

15.3.3 Interfața CallableStatement

Interfața **CallableStatement** este derivată din **PreparedStatement**, instanțele de acest tip oferind o modalitate de a apela o procedură stocată într-o bază de date, într-o manieră standar pentru toate SGBD-urile.

Crearea unui obiect **CallableStatement** se realizează prin metoda **prepareCall** a clasei **Connection**:

```
Connection con = DriverManager.getConnection(url);
CallableStatement cstmt = con.prepareCall(
    "{call proceduraStocata(?, ?)}");
```

Trimiterea parametrilor de intrare se realizează întocmai ca la **PreparedStatement**, cu metode de tip **setXXX**. Dacă procedura are și parametri de ieșire (valori returnate), aceștia vor trebui înregistrați cu metoda **registerOutParameter** înainte de execuția procedurii. Obținerea valorilor rezultate în parametrii de ieșie se va face cu metode de tip **getXXX**.

```
CallableStatement cstmt = con.prepareCall(
```

```

    "{call calculMedie(?)}");
    pstmt.registerOutParameter(1, java.sql.Types.FLOAT);
    pstmt.executeQuery();
    float medie = pstmt.getDouble(1);

```

Este posibil ca un parametru de intrare să fie și parametru de ieșire. În acest caz el trebuie să primească o valoare cu `setXXX` și, de asemenea, va fi înregistrat cu `registerOutParameter`, tipurile de date specificate trebuind să coincidă.

15.3.4 Obținerea și prelucrarea rezultatelor

15.3.5 Interfața `ResultSet`

În urma execuției unei interogări SQL rezultatul va fi reprezentat printr-un obiect de tip `ResultSet`, ce va conține toate liniile ce satisfac condițiile impuse de comanda SQL. Forma generală a unui `ResultSet` este tabelară, având un număr de coloane și de linii, funcție de secvența executată. De asemenea, obiectul va conține și meta-datele interogării cum ar fi denumirile coloanelor selectate, numărul lor, etc.

```

Statement stmt = con.createStatement();
String sql = "SELECT cod, nume FROM persoane";
ResultSet rs = stmt.executeQuery(sql);

```

Rezultatul interogării de mai sus va fi obiectul `rs` cu următoarea structură:

<u>cod</u>	<u>nume</u>
100	Ionescu
200	Popescu

Pentru a extrage informațiile din această structură va trebui să parcurgem tabelul linie cu linie și din fiecare să extragem valorile de pe coloane. Pentru acest lucru vom folosi metode de tip `getXXX`, unde `XXX` este tipul de dată al unei coloane iar argumentul primit indică fie numărul de ordine din cadrul tabelului, fie numele acestuia. Coloanele sunt numerotate de la stânga la dreapta, începând cu 1. În general, folosirea indexului coloanei în loc de numele său va fi mai eficientă. De asemenea, pentru maximă portabilitate se recomandă citirea coloanelor în ordine de la stânga la dreapta și fiecare citire să se facă o singură dată.

Un obiect `ResultSet` folosește un cursor pentru a parcurge articolele rezultate în urma unei interogări. Inițial acest cursor este poziționat înaintea primei linii, fiecare apel al metodei `next` determinând trecerea la următoarea linie. Deoarece `next` returnează `false` când nu mai sunt linii de adus, uzual va fi folosită o buclă `while-loop` pentru a itera prin articolele tabelului:

```
String sql = "SELECT cod, nume FROM persoane";
ResultSet rs = stmt.executeQuery(sql);
while (rs.next()) {
    int cod = r.getInt("cod");
    String nume = r.getString("nume");
    /* echivalent:
    int cod = r.getInt(1);
    String nume = r.getString(2);
    */
    System.out.println(cod + ", " + nume);
}
```

Implicit, un tabel de tip `ResultSet` nu poate fi modificat iar cursorul asociat nu se deplasează decât înainte, linie cu linie. Așadar, putem itera prin rezultatul unei interogări o singură dată și numai de la prima la ultima linie. Este însă posibil să creăm `ResultSet`-uri care să permită modificarea sau deplasarea în ambele sensuri. Exemplul următor va folosi un cursor care este modificabil și nu va reflecta schimbările produse de alți utilizatori după crearea sa:

```
Statement stmt = con.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_UPDATABLE);
String sql = "SELECT cod, nume FROM persoane";
ResultSet rs = stmt.executeQuery(sql);
```

Dacă un `ResultSet` folosește un cursor modificabil și care poate naviga în ambele sensuri, atunci are la dispoziție o serie de metode ce se bazează pe acest suport:

- **absolute** - Deplasează cursorul la o anumită linie specificată absolut;
- **updateXXX** - Actualizează valoarea unei coloane din linia curentă, unde *XXX* este un tip de date.

- **updateRow** - Transferă actualizările făcute liniei în baza de date.
- **moveToInsertRow** - deplasează cursorul la o linie specială, numită *linie nouă*, utilizată pentru a introduce noi articole în baza de date. Linia curentă anterioară a cursorului va fi memorată pentru a se putea reveni la ea.
- **insertRow** - inserează articolul din zona *linie nouă* în baza de date; cursorul trebuie să fie poziționat pe linia nouă la execuția acestei operațiuni.
- **moveToCurrentRow** - revine la linia curentă din tabel.
- **deleteRow** - șterge linia curentă din tabel și din baza de date; nu poate fi apelată când cursorul este în modul *linie nouă*.

Nu toate sistemele de gestiune a bazelor de date oferă suport pentru folosirea cursorilor care pot fi modificați. Pentru a determina dacă baza de date permite acest lucru pot fi utilizate metodele **supportsPositionedUpdate** și **supportsPositionedDelete** ale clasei **DatabaseMetaData**. În cazul în care acest lucru este permis, este responsabilitatea driver-ului bazei de date să asigure rezolvarea problemelor legate de actualizarea concurentă a unui cursor, astfel încât să nu apară anomalii.

15.3.6 Exemplu simplu

În continuare vom da un exemplu simplu de utilizare a claselor de bază menționate anterior. Programul va folosi o bază de date MySQL, ce conține un tabel numit *persoane*, având coloanele: *cod*, *nume* și *salariu*. Scriptul SQL de creare a bazei este:

```
create table persoane(cod integer, nume char(50), salariu double);
```

Aplicația va goli tabelul cu persoane, după care va adăuga aleator un număr de articole, va efectua afișarea lor și calculul mediei salariilor.

Listing 15.1: Exemplu simplu de utilizare JDBC

```
import java.sql.*;

public class TestJdbc {
    public static void main (String[] args) {
```

```
String url = "jdbc:mysql://localhost/test" ;
try{
    Class.forName("com.mysql.jdbc.Driver");
} catch(ClassNotFoundException e) {
    System.out.println("Eroare incarcare driver!\n" + e);
    return;
}
try{
    Connection con = DriverManager.getConnection(url);

    // Golim tabelul persoane
    String sql = "DELETE FROM persoane";
    Statement stmt = con.createStatement();
    stmt.executeUpdate(sql);

    // Adaugam un numar de persoane generate aleator
    // Tabelul persoane are coloanele (cod, nume, salariu)
    int n = 10;
    sql = "INSERT INTO persoane VALUES (?, ?, ?)";
    PreparedStatement pstmt = con.prepareStatement(sql);
    for(int i=0; i<n; i++) {
        int cod = i;
        String nume = "Persoana" + i;
        double salariu = 100 + Math.round(Math.random() *
            900);
        // salariul va fi intre 100 si 1000
        pstmt.setInt(1, cod);
        pstmt.setString(2, nume);
        pstmt.setDouble(3, salariu);
        pstmt.executeUpdate();
    }

    // Afisam persoanele ordonate dupa salariu
    sql = "SELECT * FROM persoane ORDER BY salariu";
    ResultSet rs = stmt.executeQuery(sql);
    while (rs.next())
        System.out.println(rs.getInt("cod") + ", " +
            rs.getString("nume") + ", " +
            rs.getDouble("salariu"));

    // Calculam salariul mediu
    sql = "SELECT avg(salariu) FROM persoane";
    rs = stmt.executeQuery(sql);
    rs.next();
```

```

        System.out.println("Media: " + rs.getDouble(1));

        // Inchidem conexiunea
        con.close();

    } catch(SQLException e) {
        e.printStackTrace();
    }
}
}

```

15.4 Lucrul cu meta-date

15.4.1 Interfața DatabaseMetaData

După realizarea unui conexiuni la o bază de date, putem apela metoda **get-*MetaData*** pentru a afla diverse informații legate de baza respectivă, așa numitele *meta-date* ("date despre date"); Ca rezultat al apelului metodei, vom obține un obiect de tip **DatabaseMetaData** ce oferă metode pentru determinarea tabelor, procedurilor stocate, capacităților conexiunii, gramaticii SQL suportate, etc. ale bazei de date.

Programul următor afișează numele tuturor tabelor dintr-o bază de dat înregistrată în ODBC.

Listing 15.2: Folosirea interfeței DatabaseMetaData

```

import java.sql.*;

public class TestMetaData {
    public static void main (String[] args) {
        String url = "jdbc:odbc:test";
        try{
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        } catch(ClassNotFoundException e) {
            System.out.println("Eroare incarcare driver!\n" + e);
            return;
        }
        try{
            Connection con = DriverManager.getConnection(url);
            DatabaseMetaData dbmd = con.getMetaData();
            ResultSet rs = dbmd.getTables(null, null, null, null);

```

```
        while (rs.next())
            System.out.println(rs.getString("TABLE_NAME"));
        con.close();
    } catch(SQLException e) {
        e.printStackTrace();
    }
}
}
```

15.4.2 Interfața ResultSetMetaData

Meta-datele unui `ResultSet` reprezintă informațiile despre rezultatul conținut în acel obiect cum ar fi numărul coloanelor, tipul și denumirile lor, etc. Acestea sunt obținute apelând metoda `getMetaData` pentru `ResultSet`-ul respectiv, care va returna un obiect de tip `ResultSetMetaData` ce poate fi apoi folosit pentru extragerea informațiilor dorite.

```
ResultSet rs = stmt.executeQuery("SELECT * FROM tabel");
ResultSetMetaData rsmd = rs.getMetaData();
// Aflam numarul de coloane
int n = rsmd.getColumnCount();

// Aflam numele coloanelor
String nume[] = new String[n+1];
for(int i=1; i<=n; i++)
    nume[i] = rsmd.getColumnName(i);
```


Capitolul 16

Lucrul dinamic cu clase

16.1 Incărcarea claselor în memorie

După cum știm execuția unei aplicații Java este realizată de către mașina virtuală Java (JVM), aceasta fiind responsabilă cu interpretarea codului de octeți rezultat în urma compilării. Spre deosebire de alte limbaje de programare cum ar fi C sau C++, un program Java compilat nu este descris de un fișier executabil ci de o mulțime de fișiere cu extensia `.class` corespunzătoare fiecărei clase a programului. În plus, aceste clase nu sunt încărcate toate în memorie la pornirea aplicației, ci sunt încărcate pe parcursul execuției acesteia atunci când este nevoie de ele, momentul efectiv în care se realizează acest lucru depinzând de implementarea mașinii virtuale.

Ciclu de viață al unei clase are așadar următoarele etape:

1. *Incărcarea* - Este procesul regăsirii reprezentării binare a unei clase (fișierul `.class`) pe baza numelui complet al acestuia și încărcarea acesteia în memorie. În urma acestui proces, va fi instanțiat un obiect de tip `java.lang.Class`, corespunzător clasei respective. Operațiunea de încărcare a unei clase este realizată la un moment ce precede prima utilizare efectivă a sa.
2. *Editarea de legături* - Specifică incorporarea noului tip de date în JVM pentru a putea fi utilizat.
3. *Inițializarea* - Constă în execuția blocurilor statice de inițializare și inițializarea variabilelor de clasă.

4. *Descărcarea* - Atunci când nu mai există nici o referință de tipul clasei respective, obiectul de tip `Class` creat va fi marcat pentru a fi eliminat din memorie de către *garbage collector*.

Încărcarea claselor unei aplicații Java în memorie este realizată prin intermediul unor obiecte pe care le vom numi generic *class loader*. Acestea sunt de două tipuri:

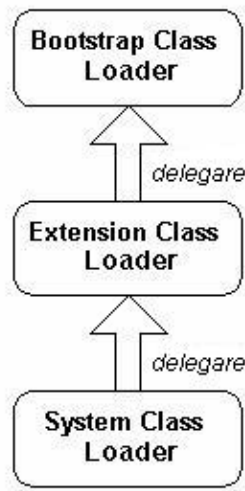
1. *Class loader-ul primordial* (*eng. bootstrap*) - Reprezintă o parte integrantă a mașinii virtuale, fiind responsabil cu încărcarea claselor standard din distribuția Java.
2. *Class loader-e proprii* - Acestea nu fac parte intrinsecă din JVM și sunt instanțe ale clasei `java.lang.ClassLoader`. Aceasta este o clasă abstractă, tipul efectiv al obiectului fiind așadar derivat din aceasta.

După cum vom vedea, la execuția unui program Java vor fi create implicit două obiecte de tip `ClassLoader` pentru încărcarea în memorie a claselor proprii ale aplicației. Există însă posibilitatea de a crea noi tipuri derivate din `ClassLoader` specializate pentru încărcarea claselor conform unor specificații anume care să realizeze diverse optimizări. Astfel, încărcarea unei clase poate determina încărcarea unor altor clase care sigur vor fi folosite împreună cu prima, sau a unor resurse ce sunt necesare funcționării acesteia, etc.

Incepând cu versiunea 1.2 de Java, a fost introdus un model de tip *delegat*, în care class loader-ele sunt dispuse ierarhic într-un arbore, rădăcina acestuia fiind class loader-ul primordial. Fiecare instanța de tip `ClassLoader` va avea așadar un *părinte* (evident, mai puțin rădăcina), acesta fiind specificat la crearea sa. În momentul când este solicitată încărcarea unei clase, un class-loader poate delega în primul rând operațiunea de încărcare părintelui său care va delega la rândul său cererea mai departe până la class loader-ul primordial sau până unul din aceștia reușește să o încarce. Abia în cazul în care nici unul din aceștia nu a reușit, va încerca să execute operațiunea de încărcare a clasei. Dacă nici ea nu va reuși, va fi aruncată o excepție de tipul **ClassNotFoundException**. Deși acest comportament nu este obligatoriu, în multe situații el este de preferat, pentru a minimiza încărcarea aceleiași clase de mai multe ori, folosind class loader-e diferite.

Implicit, Java 2 JVM oferă trei class loader-e, unul primordial și două proprii, cunoscute sub numele de:

- *Bootstrap Class Loader* - Class loader-ul primordial. Acesta este responsabil cu încărcarea claselor din distribuția Java standard (cele din pachetele `java.*`, `javax.*`, etc.).
- *Extension Class Loader* - Utilizat pentru încărcarea claselor din directoarele extensiilor JRE.
- *System Class Loader* - Acesta este responsabil cu încărcarea claselor proprii aplicațiilor Java (cele din CLASSPATH). Tipul acestuia este `java.lang.URLClassLoader`.



Intrucât tipurile de date Java pot fi încărcate folosind diverse instanțe de tip `ClassLoader`, fiecare obiect `Class` va reține class loader-ul care a fost folosit pentru încărcare, acesta putând fi obținut cu metoda `getClassLoader`.

Încărcarea *dinamică* a unei clase în memorie se referă la faptul că nu cunoaștem tipul acesteia decât la execuția programului, moment în care putem solicita încărcarea sa, specificând numele său complet prin intermediul unui șir de caractere. Acest lucru poate fi realizat prin mai multe modalități, cele mai comune metode fiind:

- **loadClass** apelată pentru un obiect de tip `ClassLoader`

```
ClassLoader loader = new MyClassLoader();
loader.loadClass("NumeCompletClasa");
```

- **Class.forName**

Această metoda va încărca respectiva clasă folosind class loader-ul obiectului curent (care o apelează):

```
Class c = Class.forName("NumeCompletClasa");
// echivalent cu
ClassLoader loader = this.getClass().getClassLoader();
loader.loadClass("ClasaNecunoscuta");

// Clasele standard pot fi si ele incarcate astfel
Class t = Class.forName("java.lang.Thread");
```

Dacă dorim să instanțiem un obiect dintr-o clasă încărcată dinamic putem folosi metoda **newInstance**, cu condiția să existe constructorul fără argumente pentru clasa respectivă. După cum vom vedea în secțiunea următoare, mai există și alte posibilități de a instanția astfel de obiecte.

```
Class c = Class.forName("java.awt.Button");
Button b = (Button) c.newInstance();
```

Folosirea interfețelor sau a claselor abstracte împreună cu încărcarea dinamică a claselor oferă un mecanism extrem de puternic de lucru în Java. Vom detalia acest lucru prin intermediul unui exemplu. Să presupunem că dorim să creăm o aplicație care să genereze aleator un vector de numere după care să aplice o anumită funcție acestui vector. Numele funcției care trebuie apelată va fi introdus de la tastatură, iar implementarea ei va fi conținută într-o clasă a directorului curent. Toate funcțiile vor extinde clasa abstractă **Funcție**. În felul acesta, aplicația poate fi extinsă cu noi funcții fără a schimba codul ei, tot ce trebuie să facem fiind să scriem noi clase care extind **Funcție** și să implementăm metoda **executa**. Aceasta va returna 0 dacă metoda s-a executat corect, -1 în caz contrar.

Listing 16.1: Exemplu de încărcare dinamică a claselor

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

public class TestFunctii {

    public static void main(String args[]) throws IOException {
        // Generam un vector aleator
        int n = 10;
        int v[] = new int[n];
        Random rand = new Random();
        for(int i=0; i<n; i++)
            v[i] = rand.nextInt(100);

        // Citim numele unei functii
        BufferedReader stdin = new BufferedReader(
            new InputStreamReader(System.in));
        String numeFunctie = "";

        while (! numeFunctie.equals("gata")) {

            System.out.print("\nFunctie:");
            numeFunctie = stdin.readLine();

            try {
                // Incarcam clasa
                Class c = Class.forName(numeFunctie);

                // Cream un obiect de tip Functie
                Functie f = (Functie) c.newInstance();

                // Setam vectorul
                f.setVector(v); // sau f.v = v;

                //Executam functia
                int ret = f.executa();
                System.out.println("\nCod returnat: " + ret);

            } catch (ClassNotFoundException e) {
                System.err.println("Functie inexistentă !");
            } catch (InstantiationException e) {
                System.err.println("Functia nu poate fi instantiata !");
            } catch (IllegalAccessException e) {
                System.err.println("Functia nu poate fi accesata !");
            }
        }
    }
}
```

```
    }  
  }  
}
```

Listing 16.2: Clasa abstractă ce descrie funcția

```
public abstract class Functie {  
    public int v[] = null;  
    public void setVector(int[] v) {  
        this.v = v;  
    }  
    public abstract int executa();  
}
```

Listing 16.3: Un exemplu de funcție

```
import java.util.*;  
public class Sort extends Functie {  
    public int executa() {  
        if (v == null)  
            return -1;  
        Arrays.sort(v);  
        for(int i=0; i<v.length; i++)  
            System.out.print(v[i] + " ");  
        return 0;  
    }  
}
```

Listing 16.4: Alt exemplu de funcție

```
public class Max extends Functie {  
    public int executa() {  
        if (v == null)  
            return -1;  
  
        int max = v[0];  
        for(int i=1; i<v.length; i++)  
            if (max < v[i])  
                max = v[i];  
        System.out.print(max);  
        return 0;  
    }  
}
```

```
}  
}
```

Un obiect de tip `URLClassLoader` menține o listă de adrese URL de unde va încerca să încarce în memorie clasa al cărei nume îl specificăm ca argument al metodelor de mai sus. Implicit, la crearea class loader-ului această listă este completată cu informațiile din variabila sistem `CLASSPATH` sau cu cele specificate prin opțiunea `-classpath` la lansarea aplicației. Folosind metoda `getURLs` putem afla aceste adrese, iar cu `addURL` putem adăuga o nouă adresă de căutare a claselor. Bineînțeles, adresele URL pot specifica și directoare ale sistemului de fișiere local. Să presupunem că în directorul `c:\clase\demo` există clasa cu numele `Test`, aflată în pachetul `demo` și dorim să o încărcăm dinamic în memorie:

```
// Obținem class loaderul curent  
URLClassLoader urlLoader =  
    (URLClassLoader) this.getClass().getClassLoader();  
// Adaugam directorul sub forma unui URL  
urlLoader.addURL(new File("c:\\clase").toURL());  
// Incarcam clasa  
urlLoader.loadClass("demo.Test");
```

După ce o clasă a fost încărcată folosind un class loader, ea nu va mai putea fi descărcată explicit din memorie. În cazul în care dorim să avem posibilitatea de a o reîncărca, deoarece a fost modificată și recompilată, trebuie să folosim class-loadere proprii și să instanțiem noi obiecte de tip `ClassLoader`, ori de câte ori dorim să forțăm reîncărcarea claselor. Crearea unui class loader propriu se face uzual prin extinderea clasei `URLClassLoader`, o variantă simplistă fiind prezentată mai jos:

```
public class MyClassLoader extends URLClassLoader{  
    public MyClassLoader(URL[] urls){  
        super(urls);  
    }  
}
```

Încărcarea claselor folosind clasa nou creată se va face astfel:

```
// La initializare
URLClassLoader systemLoader =
    (URLClassLoader) this.getClass().getClassLoader();
URL[] urls = systemLoader.getURLs();

// Cream class loaderul propriu
MyClassLoader myLoader = new MyClassLoader(urls);
myLoader.loadClass("Clasa");
...
// Dorim sa reincarcam clasa
myLoader.loadClass("Clasa"); // nu functioneaza !

// Cream alt class loader
MyClassLoader myLoader = new MyClassLoader(urls);
myLoader.loadClass("Clasa"); // reincarca clasa
```

16.2 Mecanismul reflectării

Mecanismul prin care o clasă, interfață sau obiect "reflectă" la momentul execuției structura lor internă se numește *reflectare* (eng. reflection), acesta punând la dispoziție metode pentru:

- Determinarea clasei unui obiect.
- Aflarea unor informații despre o clasă (modificatori, superclasa, constructori, metode).
- Instanțierea unor clase al căror nume este știut abia la execuție.
- Setarea sau aflarea atributelor unui obiect, chiar dacă numele acestora este știut abia la execuție.
- Invocarea metodelor unui obiect al căror nume este știut abia la execuție.
- Crearea unor vectori a căror dimensiune și tip nu este știut decât la execuție.

Suportul pentru reflectare este inclus în distribuția standard Java, fiind cunoscut sub numele de *Reflection API* și conține următoarele clase:

- `java.lang.Class`
- `java.lang.Object`
- Clasele din pachetul `java.lang.reflect` și anume:
 - `Array`
 - `Constructor`
 - `Field`
 - `Method`
 - `Modifier`

16.2.1 Examinarea claselor și interfețelor

Examinarea claselor și interfețelor se realizează cu metode ale clasei `java.lang.Class`, un obiect de acest tip putând să reprezinte atât o clasă cât și o interfață, diferențierea acestora făcându-se prin intermediul metodei `isInterface`.

Reflection API pune la dispoziție metode pentru obținerea următoarelor informații:

Aflarea instanței `Class` corespunzător unui anumit obiect sau tip de date:

```
Class c = obiect.getClass();
Class c = java.awt.Button.class;
Class c = Class.forName("NumeClasa");
```

Tipurile primitive sunt descrise și ele de instanțe de tip `Class` având forma *TipPrimitiv.class*: `int.class`, `double.class`, etc., diferențierea lor făcându-se cu ajutorul metodei `isPrimitive`.

Aflarea numelui unei clase - Se realizează cu metoda `getName`:

```
Class clasa = obiect.getClass();
String nume = clasa.getName();
```

Aflarea modificatorilor unei clase - Se realizează cu metoda `getModifiers`, aceasta returnând un număr întreg ce codifică toți modificatorii clasei. Pentru a determina ușor prezența unui anumit modificador se folosesc metodele statice ale clasei `Modifier` `isPublic`, `isAbstract` și `isFinal`:

```
Class clasa = obiect.getClass();
int m = clasa.getModifiers();
String modif = "";
if (Modifier.isPublic(m))
    modif += "public ";
if (Modifier.isAbstract(m))
    modif += "abstract ";
if (Modifier.isFinal(m))
    modif += "final ";
System.out.println(modif + "class" + c.getName());
```

Aflarea superclasei - Se realizează cu metoda `getSuperclass` ce returnează o instanță de tip `Class`, corespunzătoare tipului de date al superclasei sau `null` pentru clasa `Object`.

```
Class c = java.awt.Frame.class;
Class s = c.getSuperclass();
System.out.println(s); // java.awt.Window

Class c = java.awt.Object.class;
Class s = c.getSuperclass(); // null
```

Aflarea interfețelor implementate de o clasă sau extinse de o interfață - Se realizează cu metoda `getInterfaces`, ce returnează un vector de tip `Class[]`.

```
public void interfete(Class c) {
    Class[] interf = c.getInterfaces();
    for (int i = 0; i < interf.length; i++) {
        String nume = interf[i].getName();
        System.out.print(nume + " ");
    }
}
```

```

    }
}
...
interfete(java.util.HashSet.class);
// Va afisa interfetele implementate de HashSet:
// Cloneable, Collection, Serializable, Set

interfete(java.util.Set);
// Va afisa interfetele extinse de Set:
// Collection

```

Aflarea variabilelor membre - Se realizează cu una din metodele `getFields` sau `getDeclaredFields`, ce returnează un vector de tip `Field[]`, diferența între cele două constând în faptul că prima returnează *toate* variabilele membre, inclusiv cele moștenite, în timp ce a doua le returnează doar pe cele declarate în cadrul clasei. La rândul ei, clasa `Field` pune la dispoziție metodele `getName`, `getType` și `getModifiers` pentru a obține numele, tipul, respectiv modificatorii unei variabile membru.

Cu ajutorul metodei `getField` este posibilă obținerea unei referințe la o variabilă membră cu un anumit nume specificat.

Aflarea constructorilor - Se realizează cu metodele `getConstructors` sau `getDeclaredConstructors`, ce returnează un vector de tip `Constructor[]`. Clasa `Constructor` pune la dispoziție metodele `getName`, `getParameterTypes`, `getModifiers`, `getExceptionTypes` pentru a obține toate informațiile legate de respectivul constructor.

Cu ajutorul metodei `getConstructor` este posibilă obținerea unei referințe la constructor cu o semnătură specificată.

Aflarea metodelor - Se realizează cu metodele `getMethods` sau `getDeclaredMethods`, ce returnează un vector de tip `Method[]`. Clasa `Method` pune la dispoziție metodele `getName`, `getParameterTypes`, `getModifiers`, `getExceptionTypes`, `getReturnType` pentru a obține toate informațiile legate

de respectiva metodă.

Cu ajutorul metodei `getMethod` este posibilă obținerea unei referințe la o metodă cu o semnătură specificată.

Aflarea claselor imbricate - Se realizează cu metodele `getClasses` sau `getDeclaredClasses`, ce returnează un vector de tip `Class []`.

Aflarea clasei de acoperire - Se realizează cu metoda `getDeclaringClass`. Această metodă o regăsim și în clasele `Field`, `Constructor`, `Method`, pentru acestea returnând clasa cărei îi aparține variabila, constructorul sau metoda respectivă.

16.2.2 Manipularea obiectelor

Pe lângă posibilitatea de a examina structura unei anumite clase la momentul execuției, folosind *Reflection API* avem posibilitatea de a lucra dinamic cu obiecte, bazându-ne pe informații pe care le obținem abia la execuție.

Crearea obiectelor

După cum știm, crearea obiectelor se realizează cu operatorul `new` urmat de un apel la un constructor al clasei pe care o instanțiem. În cazul în care numele clasei nu este cunoscut decât la momentul execuției nu mai putem folosi această metodă de instanțiere. În schimb, avem la dispoziție alte două variante:

- Metoda `newInstance` din clasa `java.lang.Class`
Aceasta permite instanțierea unui obiect folosind constructorul fără argumente al acestuia. Dacă nu există un astfel de constructor sau nu este accesibil vor fi generate excepții de tipul `InstantiationException`, respectiv `IllegalAccessException`.

```
Class c = Class.forName("NumeClasa");  
Object o = c.newInstance();  
  
// Daca stim tipul obiectului
```

```
Class c = java.awt.Point.class;
Point p = (Point) c.newInstance();
```

- Metoda **newInstance** din clasa **Constructor**

Aceasta permite instanțierea unui obiect folosind un anumit constructor, cel pentru care se face apelul. Evident, această soluție presupune în primul rând obținerea unui obiect de tip **Constructor** cu o anumită semnătură, apoi specificarea argumentelor la apelarea sa. Să rescriem exemplul de mai sus, apelând constructorul cu două argumente al clasei **Point**.

```
Class clasa = java.awt.Point.class;

// Obținem constructorul dorit
Class[] semnatura = new Class[] {int.class, int.class};
Constructor ctor = clasa.getConstructor(semnatura);

// Pregătim argumentele
// Ele trebuie să fie de tipul referință corespunzător
Integer x = new Integer(10);
Integer y = new Integer(20);
Object[] arg = new Object[] {x, y};

// Instanțiem
Point p = (Point) ctor.newInstance(arg);
```

Excepții generate de metoda **newInstance** sunt: **InstantiationException**, **IllegalAccessException**, **IllegalArgumentException** și **InvocationTargetException**. Metoda **getConstructor** poate provoca excepții de tipul **NoSuchMethodException**.

Invocarea metodelor

Invocarea unei metode al cărei nume îl cunoaștem abia la momentul execuției se realizează cu metoda **invoke** a clasei **Method**. Ca și în cazul constructorilor, trebuie să obținem întâi o referință la metoda cu semnătura corespunzătoare și apoi să specificăm argumentele. În plus, mai putem obține valoarea returnată. Să presupunem că dorim să apelăm metoda **contains**

a clasei `Rectangle` care determină dacă un anumit punct se găsește în interiorul dreptunghiului. Metoda `contains` are mai multe variante, noi o vom apela pe cea care acceptă un argument de tip `Point`.

```
Class clasa = java.awt.Rectangle.class;
Rectangle obiect = new Rectangle(0, 0, 100, 100);

// Obținem metoda dorita
Class[] signatura = new Class[] {Point.class};
Method metoda = clasa.getMethod("contains", signatura);

// Pregătim argumentele
Point p = new Point(10, 20);
Object[] arg = new Object[] {p};

// Apelăm metoda
metoda.invoke(obiect, arg);
```

Dacă numărul argumentelor metodei este 0, atunci putem folosi valoarea `null` în locul vectorilor ce reprezintă signatura, respectiv parametri de apelare ai metodei.

Excepțiile generate de metoda `invoke` sunt: `IllegalAccessException` și `InvocationTargetException`. Metoda `getMethod` poate provoca excepții de tipul `NoSuchMethodException`.

Setarea și aflarea variabilelor membre

Pentru setarea și aflarea valorilor variabilelor membre sunt folosite metodele `set` și `get` ale clasei `Field`. Să presupunem că dorim să setăm variabila `x` a unui obiect de tip `Point` și să obținem valoarea variabilei `y` a aceluiași obiect:

```
Class clasa = java.awt.Point.class;
Point obiect = new Point(0, 20);

// Obținem variabilele membre
Field x, y;
x = clasa.getField("x");
y = clasa.getField("y");
```

```
// Setam valoarea lui x
x.set(obiect, new Integer(10));

// Obtinem valoarea lui y
Integer val = y.get(obiect);
```

Excepțiile generate de metodele `get`, `set` sunt: `IllegalAccessException` și `IllegalArgumentException`. Metoda `getField` poate provoca excepții de tipul `NoSuchFieldException`.

Revenind la exemplul din secțiunea anterioară cu apelarea dinamică a unor funcții pentru un vector, să presupunem că există deja un număr însemnat de clase care descriu diferite funcții dar acestea nu extind clasa abstractă `Funcție`. Din acest motiv, soluția anterioară nu mai este viabilă și trebuie să folosim apelarea metodei `executa` într-un mod dinamic.

Listing 16.5: Lucru dinamic cu metode și variabile

```
import java.lang.reflect.*;
import java.util.*;
import java.io.*;

public class TestFuncții2 {

    public static void main(String args[]) throws IOException {
        // Generam un vector aleator
        int n = 10;
        int v[] = new int[n];
        Random rand = new Random();
        for(int i=0; i<n; i++)
            v[i] = rand.nextInt(100);

        // Citim numele unei funcții
        BufferedReader stdin = new BufferedReader(
            new InputStreamReader(System.in));
        String numeFuncție = "";

        while (! numeFuncție.equals("gata")) {

            System.out.print("\nFuncție:");
            numeFuncție = stdin.readLine();
```

```

try {
    // Incarcam clasa
    Class c = Class.forName( numeFunctie );

    // Cream un obiect de tip Functie
    Object f = c.newInstance();

    // Setam vectorul (setam direct variabila v)
    Field vector = c.getField("v");
    vector.set(f, v);

    // Apelam metoda 'executa'
    // Folosim null pentru ca nu avem argumente
    Method m = c.getMethod("executa", null);
    Integer ret = (Integer) m.invoke(f, null);
    System.out.println("\nCod returnat: " + ret);

} catch (Exception e) {
    System.err.println("Eroare la apelarea functiei !");
}
}
}
}
}

```

16.2.3 Lucrul dinamic cu vectori

Vectorii sunt reprezentați ca tip de date tot prin intermediul clasei `java.lang.Class`, diferențierea făcându-se prin intermediul metodei `isArray`.

Tipul de date al elementelor din care este format vectorul va fi obținut cu ajutorul metodei `getComponentType`, ce întoarce o referință de tip `Class`.

```

Point []vector = new Point[10];
Class c = vector.getClass();
System.out.println(c.getComponentType());
// Va afisa: class java.awt.Point

```

Lucrul dinamic cu obiecte ce reprezintă vectori se realizează prin intermediul clasei **Array**. Aceasta conține o serie de metode statice ce permit:

- Crearea de noi vectori: `newInstance`
- Aflarea numărului de elemente: `getLength`

- Setarea / aflarea elementelor: `set`, `get`

Exemplul de mai jos creează un vector ce conține numerele întregi de la 0 la 9:

```
Object a = Array.newInstance(int.class, 10);
for (int i=0; i < Array.getLength(a); i++)
    Array.set(a, i, new Integer(i));

for (int i=0; i < Array.getLength(a); i++)
    System.out.print(Array.get(a, i) + " ");
```