CUPRINS

6. Clase abstracte si interfete

7. Colectii de obiecte în Java

8. Reutilizarea codului în POO

9. Clase incluse

10. Clase pentru o interfatã graficã

11. Programare bazatã pe evenimente

1. Java ca limbaj de programare cu obiecte

Diferente între limbajele Java si C

Limbajul Java foloseste aceleasi instructiuni cu limbajul C, mai putin instructiunea *goto*. Tipurile de date primitive sunt aproape aceleasi, plus tipul *boolean*, care a schimbat putin si sintaxa unor instructiuni. Diferentele importante apar la tipurile de date derivate (vectori, structuri, pointeri) si la structura programelor.

In limbajul C existã un singur fel de comentarii, care încep prin perechea de caractere "/*" si se terminã prin perechea de caractere "*/".In C++ au apãrut, în plus, comentarii care încep prin perechea de caractere "//" si se terminã la sfârsitul liniei în care apare acel comentariu. Java preia aceste douã feluri de comentarii, la care se adaugã comentarii destinate generãrii automate a documentatiilor programelor (cu ajutorul programului "javadoc"); aceste comentarii încep printr-un grup de 3 caractere "/**" si se terminã la fel cu comentariile C, prin "*/"

Exemplu:

```
/**  Clasa  Heap
* @ Data : Apr. 2000
*/
```

Tipurile de date primitive

Java preia de la C si C++ aproape toate tipurile aritmetice (*short, int, long, float, double*) si tipul *void*, dar impune o aceeasi lungime si reprezentare a tipurilor numerice pentru toate implementãrile limbajului. Un întreg de tip *int* ocupã 32 de biti, un *short* ocupã 16 biti, iar un *long* ocupã 64 de biti. Un *float* ocupã 32 de biti iar un *double* ocupa 64 de biti. Tipul aritmetic *byte* ocupã 8 biti (valori între –128 si 127).

Tipul *char* ocupã 16 biti pentru cã standardul de reprezentare a caracterelor este UTF-16 sau Unicode (în locul codului ASCII) si permite utilizarea oricãrui alfabet.

Toate tipurile aritmetice din Java reprezintã numere cu semn si nu mai existã cuvântul *unsigned* pentru declararea de variabile aritmetice fãrã semn.

Tipul *boolean* din Java ocupã un singur bit; constantele de tip *boolean* sunt *true* si *false*. Existenta acestui tip modificã si sintaxa instructiunilor *if, while, do* si a expresiei conditionale, precum si rezultatul expresiilor de relatie (care este acum de tip *boolean* si nu de tip *int*). Asadar, instructiunile urmãtoare sunt gresite sintactic în Java, desi sunt corecte în C si C++.

```
while ( d++ = s++) ;          // corect este : while ( (d++=s++) !=0) ;
return x ? 1:0 ;                  // corect este:  return  x !=0 ? 1:0 ;  cu x de tip "int"
if ( ! n)  { ...  }                  // corect este:  if (n==0) { ... }
do { nf=nf *n--;} while (n) ;     // corect este:  do { nf=nf*n--;} while ( n>0);
```

Variabilele declarate în functii nu primesc valori implicite iar compilatorul semnaleazã utilizarea de variabile neinitializate explicit de programator.

In Java, se fac automat la atribuire numai conversiile de “promovare” de la un tip numeric “inferior” la un tip aritmetic “superior”, care nu implicã o trunchiere. Exemple:

```
int n=3; float f; double d;
d=f=n;                  // corect  f=3.0, d=3.0
n=f;                    // gresit sintactic
f=d;                    // gresit sintactic
```

Ierarhizarea tipurilor aritmetice, de la “inferior” la “superior” este:
byte, short, int, long, float, double
Tipul *char* nu este un tip aritmetic dar se pot face conversii prin operatorul (tip) între tipul char si orice tip aritmetic întreg. Exemplu:

```
byte b=65;    char ch;  ch =(char)b;        // ch este 'A'
ch='\n';  b =(byte)ch;                      // b este 10
```

Aceleasi reguli de conversie între tipuri numerice se aplicã si între argumentele efective si argumentele formale, deoarece compilatorul face automat o atribuire a valorii argumentului efectiv la argumentul formal corespunzãtor. Exemplu:

```
double  r = Math.sqrt(2);       // promovare de la int la double ptr. 2
```

O altã conversie automatã, de promovare se face pentru rezultatul unei functii, dacã tipul expresiei din instructiunea *return* diferã de tipul declarat al functiei. Exemplu:

```
static float rest (float a, float b) {
  int r = (int)a % (int) b;
  return r;
}
```

Conversia de la un tip numeric “superior” la un tip aritmetic “inferior” trebuie cerutã explicit prin folosirea operatorului “cast” de fortare a tipului si nu se face automat ca în C. Exemple :

```
f= (float)d;                    // cu pierdere de precizie
n=(int)f;                       // cu trunchiere
int r = (int) Math.sqrt(4);     // conversie necesara de la double la int

   // functie de rotunjire din clasa Math
public static int round (float a) {
  return (int)floor(a + 0.5f);        // "floor" are rezultat double
}
```

Compilatorul Java verificã dacã este specificat un rezultat la orice iesire posibilã dintr-o functie si nu permite instructiuni *if* fãrã *else* în functii cu tip diferit de *void*.

In Java nu există operatorul *sizeof* din C, pentru determinarea memoriei ocupate de un tip sau de o variabilă, pentru că nu este necesar acest operator.

Cea mai importantă diferentă dintre Java, pe de o parte, si limbajele C, C++ pe de altă parte, este absenta tipurilor pointer din Java. Deci nu există posibilitatea de a declara explicit variabile pointer si nici operatorii unari '&' (pentru obtinerea adresei unei variabile) si '*' (indirectare printr-un pointer). Operatorul *new* pentru alocare dinamică din C++ există în Java, dar are ca rezultat o referintă si nu un pointer.

Supradefinirea functiilor

Supradefinirea sau supraîncărcarea functiilor ("Function Overloading") a fost introdusă în C++ pentru a permite definirea mai multor functii cu acelasi nume si cu acelasi tip dar cu argumente diferite într-o aceeasi clasă. Pot exista functii cu acelasi nume (eventual si cu acelasi argumente si tip) în clase diferite, dar acesta nu este un caz de supradefinire, fiindcă ele se află în spatii de nume diferite.

In Java, ca si în C++, o functie este deosebită de alte functii din aceeasi clasă (de către compilator) prin "semnătura" sa (prin "amprenta" functiei), care este formată din numele, tipul si argumentele functiei. Un exemplu uzual de functii supradefinite este cel al functiilor de afisare la consolă în mod text "print" si "println", care au mai multe definitii, pentru fiecare tip de date primitiv si pentru tipurile clasă *String* si *Object* :

```
        // din pachetul java.io
public class PrintStream ...{          // este o clasă derivată
  public void print (int i) {          // scrie un întreg
    write (String.valueOf(i));
  }
  public void print (float f) {        // scrie un număr real
    write (String.valueOf(f));
  }
  public void print (boolean b) {      // scrie un boolean
    write (b ? "true" : "false");
  }
  public void print (String s) {       // scrie un sir de caractere
    if (s== null)  s= "null";
    write (s);
  }
```

Functia "String.valueOf" este si ea supradefinită pentru diferite argumente.

Declaratii de variabile

O declaratie de variabilă poate să apară fie într-o functie, fie în afara functiilor, dar într-o clasă; nu există variabile externe claselor. Locul declaratiei este important : o variabilă dintr-o functie este locală acelei functii, iar o variabilă declarată la nivel de clasă este utilizabilă de orice functie din clasă (si chiar de functii din alte clase).

In C toate declaratiile dintr-un bloc trebuie să preceadă prima instructiune executabilă din acel bloc. In C++ si în Java o declaratie poate apare oriunde într-un

bloc, între alte instructiuni sau declaratii. Domeniul de valabilitate al unei variabile începe în momentul declarãrii si se terminã la sfârsitul blocului ce contine declaratia.

Instructiunea *for* constituie un caz special: variabila contor se declarã de obicei în instructiunea *for*, iar valabilitatea acestei declaratii este limitatã la instructiunile repetate prin instructiunea *for* . Exemplu:

```
public static boolean  este ( int x[ ], int y) {
int n=x.length;                          // lungime vector x
for (int k=0; k<n; k++)
  if ( x[k]==y)
     break;
return  k==n ? false : true;             // eroare:   k nedeclarat !
}
```

In Java nu existã cuvântul cheie *const* iar constantele sunt declarate ca variabile cu atributele *static* si *final*. Exemplu:

```
public static final double PI = 3.14159265358979323846;       // in clasa Math
```

Alte diferente între Java si C

In Java nu existã declaratia *typedef* deoarece definirea unei clase introduce automat un nume pentru un nou tip de date.

In Java nu existã operatorul *sizeof* , pentru cã lungimea variabilelor este cunoscutã, iar la alocarea de memorie (cu *new*) nu trebuie specificatã dimensiunea alocatã.

Compilatorul Java nu are preprocesor, deci nu existã directivele preprocesor atât de mult folosite în C si în C++ : *#define, #include,* etc.

Structura programelor Java

O aplicatie Java contine cel putin o clasã, care contine cel putin o metodã cu numele "main" de tip *void* si cu atributele *static* si *public*. Metoda "main" trebuie sã aibã ca unic argument un vector de obiecte *String*. Ca si în C, executia unui program începe cu functia “main”, doar cã “main” trebuie sã fie inclusã, ca metodã staticã, într-o clasã si trebuie sã aibã un argument vector de siruri.

Exemplul urmãtor este un program minimal, care afiseazã un text constant:

```
public class Main {
  public static void main (String arg[) {
     System.out.println (" Main started  ");
  }
}
```

In Java nu conteazã ordinea în care sunt scrise functiile (metodele) unei clase, deci o functie poate fi apelatã înainte de a fi definitã si nici nu este necesarã declararea functiilor utilizate (nu se folosesc prototipuri de functii). Orice functie apartine unei clase si nu se pot defini functii în afara claselor.

In exemplele urmãtoare se vor folosi numai clase care reunesc câteva metode statice, functii care pot fi executate fãrã a crea obiecte de tipul clasei respective.

Exemplul urmãtor este un fisier sursã cu o singurã clasã, care contine douã metode, ambele publice si statice:

```
class Main {
  public static void main (String arg[ ]) {          // cu "main" incepe executia
      writeln ("Hello world !");
  }
  public static void writeln (String txt) {          // afiseaza un text pe ecran
       System.out.println (txt);
   }
}
```

Numele unei metode statice trebuie precedat de numele clasei din care face parte (separate printr-un punct), dacã este apelatã dintr-o metodã a unei alte clase. Exemplu:

```
public class Main {
   public static void main (String arg[ ]) {         // cu "main" incepe executia
      Util.writeln ("Hello world !");
   }
}
public class Util {
  public static void writeln (String txt) {          // afiseaza un text pe ecran
       System.out.println (txt);
   }
}
```

O metodã ne-staticã trebuie apelatã pentru un anumit obiect, iar numele ei trebuie precedat de numele obiectului (si un punct). Metoda "println" este apelatã pentru obiectul adresat de variabila "out", variabilã publicã din clasa *System*.

Un fisier sursã Java poate contine mai multe clase, dar numai una din ele poate avea atributul *public*. Numele fisierului sursã (de tip “java”) trebuie sã coincidã cu numele clasei publice pe care o contine. O clasã publicã este accesibilã si unor clase din alte pachete de clase.

Compilatorul Java creeazã pentru fiecare clasã din fisierul sursã câte un fisier cu extensia “class” si cu numele clasei. Dacã este necesar, se compileazã si alte fisiere sursã cu clase folosite de fisierul transmis spre compilare.

Faza de executie a unui program Java constã din încãrcarea si interpretarea tuturor claselor necesare executiei metodei “main” din clasa specificatã în comanda “java”.

Tipuri clasã si tipuri referintã

O clasã este o structurã care poate contine atât date cât si functii ca membri ai structurii. In Java nu mai existã cuvântul cheie *struct* , iar definirea unei clase foloseste cuvântul cheie *class*. Se pot defini clase ce contin numai date publice, echivalente structurilor din limbajele C si C++. Exemplu:

```
public class Point {          // orice punct din plan
   public double x, y;        // coordonate punct
}
   // creare si utilizare obiect din clasa "Point"
Point a = new Point();          // constructor implicit, generat automat
a.x=2.; a.y =-3;                // un punct in cadranul 4
```

In practicã se preferã ca variabilele clasei "Point" sã fie de tip *private* (inaccesibile unor metode din alte clase) si ca initializarea lor sã se facã în constructorul clasei:

```
public class Point {                          // orice punct din plan
   private double x,y;                        // coordonate punct
   public Point (double xi, double yi) {      // functie constructor
     x=xi; y=yi;
   }
}
```

Clasele (neabstracte) Java sunt de douã categorii:
- Clase instantiabile, care pot genera obiecte, care contin date si metode (ne-statice).
- Clase neinstantiabile, care contin doar metode statice (si eventual constante).

O metodã staticã corespunde unei functii din limbajul C, cu diferenta cã numele functiei trebuie precedat de numele clasei din care face parte. Exemple:

```
double xabs = Math.abs(x);       // valoarea absoluta a lui x
double y = Math.sqrt(x);         // radical din x
int n = Integer.parseInt (str);  // conversie sir "str" la tipul "int"
```

Definirea unei clase instantiabile T creeazã automat un nou tip de date T. Un obiect de tip T este o instantiere a clasei T si este referit printr-o variabilã de tip T. Clasa Java cu numele *String* defineste un tip de date *String*, ce poate fi folosit în declararea de variabile, vectori sau functii de tip *String*. Exemple:

```
String msg = "Eroare";                  // o variabila sir
String tcuv[] ={"unu","doi","trei"};    // un vector de siruri
```

Un obiect Java corespunde unei variabile structurã din C, iar o variabilã de un tip clasã corespunde unei variabile pointer la o structurã din C.

In Java toate obiectele sunt alocate dinamic, folosind operatorul *new*, iar variabila de tip clasã trebuie initializatã cu rezultatul operatorului *new*. Exemplu:

```
String  mesaj;                          // String este o clasã predefinitã
mesaj = new String (“ Eroare ! “) ;   // alocare memorie pentru sir
```

Pentru constantele de tip *String* se creeazã obiecte automat, de cãtre compilator, ale cãror adrese pot fi folosite în atribuiri sau initializãri la declarare. Exemple:

```
System.out.println ("Eroare !");
String msg;  msg = " Corect";
```

Clasa *String* contine mai multe metode (functii) publice, utilizabile în alte clase. De exemplu, metoda *length*, fãrã argumente, are ca rezultat (întreg) lungimea sirului continut în obiectul de tip *String* pentru care se apeleazã metoda. Exemplu:

```
int  len = mesaj.length();
```

Acest exemplu aratã cã membrii unei clase se folosesc la fel cu membrii unei structuri, indiferent cã ei sunt variabile (câmpuri) sau functii (metode).

Un alt exemplu este o constructie mult folositã în Java pentru afisarea la consolã (în mod text) a unor siruri de caractere:

```
System.out.println (mesaj);
System.out.println ( " Eroare ");
```

In aceste exemple *System* este numele unei clase predefinite, *out* este numele unei variabile publice (din clasa *System*) de un tip clasã (*PrintStream*), iar *println* este numele unei metode din clasa *PrintStream*.

Numele unei metode poate fi precedat de numele unei variabile clasã sau de numele unei clase, dar întotdeauna caracterul separator este un punct. Este uzual în Java sã avem denumiri de variabile sau de metode care contin câteva puncte de separare a numelor folosite în precizarea contextului. Exemple:

```
if ( Character.isDigit ( str.charAt(0)) ) . . .    // daca primul caracter e o cifra
System.out.println (obj + obj.getClass().getName());
int maxdigits= (Integer.MAX_VALUE+"").length();
```

O referintã la un tip clasã T este de fapt un pointer la tipul T dar care se foloseste ca si cum ar fi o variabilã de tipul T. Indirectarea prin variabila referintã este realizatã automat de compilator, fãrã a folosi un operator special, ca în C .

Tipul referintã a fost introdus în C++ în principal pentru a declara parametri modificabili în functii, cu simplificarea scrierii si utilizãrii acestor functii.

In Java nu trebuie folositã o sintaxã specialã pentru declararea de variabile sau de parametri referintã, deoarece toate variabilele de un tip clasã sunt automat considerate ca variabile referintã. Nu se pot defini referinte la tipuri primitive.

O variabilã referintã Java nu este un obiect, dar contine adresa unui obiect alocat dinamic. O variabilã referintã apare de obicei în stânga unei atribuiri cu operatorul *new* sau cu constanta *null* în partea dreaptã. Exemplu:

```
Vector a = new Vector( );       // a = variabila referinta la un obiect de tip Vector
```

Atunci când se apeleazã o metodã pentru un obiect, se foloseste numele variabilei referintã ca si cum acest nume ar reprezenta chiar obiectul respectiv si nu adresa sa:

```
System.out.println ( a.size() );          // afisare dimensiune vector a
```

Operatorul de concatenare '+', folosit între obiecte de tip *String*, poate crea impresia cã variabilele de tip *String* contin chiar sirurile care se concateneazã si nu adresele lor. Exemple:

```
String s1="java.", s2="util.", s3="Rand";
System.out.println (s1+s2+s3);                  // scrie java.util.Rand
```

Operatorul de concatenare “+” este singurul operator “supradefinit” în Java si el poate fi utilizat între operanzi de tip *String* sau cu un operand de tip *String* si un alt operand de orice tip primitiv sau de un tip clasã (pentru care existã o functie de conversie la tipul *String*). Exemplu:

```
int a=3, b=2 ;
System.out.println ( a + “+” + b + “=“ + (a+b));    // scrie:  3 + 2 = 5
```

Efectul operatorului '+' depinde de tipul operanzilor: dacã unul din operanzi este de tip *String* atunci este interpretat ca operator de concatenare iar rezultatul este tot *String*.

Spatii ale numelor în Java

Un spatiu al numelor ("namespace") este un domeniu de valabilitate pentru un nume simbolic ales de programator. In cadrul unui spatiu nu pot exista douã sau mai multe nume identice (exceptie fac metodele supradefinite dintr-o aceeasi clasã). Pot exista nume identice în spatii diferite.

Fiecare clasã creeazã un spatiu de nume pentru variabilele si metodele clasei; ca urmare numele metodelor sau datelor publice dintr-o clasã trebuie precedate de numele clasei, atunci când se folosesc în alte clase. Exemple:

```
Main.writeln ("abc");       // clasa Main, metoda writeln
Math.sqrt(x);               // clasa Math, metoda sqrt
System.out                  // clasa System, variabila out
```

Clasele înrudite ca rol sau care se apeleazã între ele sunt grupate în "pachete" de clase ("package"). Instructiunea *package* se foloseste pentru a specifica numele pachetului din care vor face parte clasele definite în fisierul respectiv; ea trebuie sã fie prima instructiune din fisierul sursã Java. In lipsa unei instructiuni *package* se considerã cã este vorba de un pachet anonim implicit, situatia unor mici programe de test pentru depanarea unor clase. Un nume de pachet corespunde unui nume de director, în care sunt grupate fisierele .class corespunzãtoare claselor din pachet.

Numele unui pachet poate avea mai multe componente, separate prin puncte. Numele de pachete cu clase predefinite, parte din JDK, încep prin *java* sau *javax*. Exemple :

```
java.io , java.util.regex , java.awt, javax.swing.tree
```

Un pachet este un spatiu al numelor pentru numele claselor din acel pachet.

In general numele unei clase (publice) trebuie precedat de numele pachetului din care face parte, atunci când este folosit într-un alt pachet.

De observat cã un fisier sursã nu creeazã un spatiu de nume; este posibil si chiar uzual ca în componenta unui pachet sã intre clase aflate în fisiere sursã diferite, dar care au la început aceeasi instructiune "package".

Pachetul cu numele "java.lang" ("language") este folosit de orice program Java si de aceea numele lui nu mai trebuie mentionat înaintea numelui unei clase din *java.lang*. De exemplu clasa *String* face parte din pachetul java.lang.

Exemplu care ilustreazã utilizarea unei clase dintr-un alt pachet decât java.lang :

```
public static void main (String arg[]) {
    java.util.Random rand =new java.util.Random();
    for (int i=1;i<=10;i++)                    // scrie 10 numere aleatoare
      System.out.println ( rand.nextFloat());
 }
```

Variabila cu numele "rand" este de tipul *Random*, iar clasa *Random* este definitã în pachetul "java.util". Notatia "rand.nextFloat()" exprimã apelul metodei "nextFloat" din clasa "Random" pentru obiectul adresat de variabila "rand".

Instructiunea *import* permite simplificarea referirilor la clase din alte pachete si poate avea mai multe forme. Cea mai folositã formã este:

```
import pachet.* ;
```

Instructiunea anterioarã permite folosirea numelor tuturor claselor dintr-un pachet cu numele "pachet", fãrã a mai fi precedate de numele pachetului. Exemplul urmãtor ilustreazã folosirea instructiunii "import":

```
import java.util.*;          // sau    import  java.util.Random;
class R {
  public static void main (String arg[]) {
     Random rand =new Random();
     for (int i=1;i<=10;i++)                    // scrie 10 numere aleatoare
       System.out.println ( rand.nextFloat());
  }
}
```

Uneori se preferã importul de clase individuale, atât pentru documentare cât si pentru evitarea ambiguitãtilor create de clase cu acelasi nume din pachete diferite. Exemplu care aratã riscurile importului tuturor claselor dintr-un pachet:

```
import java.util.*;
import java.awt.*;
class test {
  public static void main (String av[ ]) {
     List list; . . .   // clasa  java.awt.List sau interfata  java.util.List ?
  }
}
```

Definirea si utilizarea de vectori în Java

Cuvântul “vector” este folosit aici ca echivalent pentru “array” din limba englezã si se referã la un tip de date implicit limbajelor C, C++ si Java. Acest tip este diferit de tipul definit de clasa JDK *Vector* (vectori ce se pot extinde automat) si de aceea vom folosi si denumirea de vector intrinsec (limbajului) pentru vectori ca cei din C.

In Java, declararea unei variabile (sau unui parametru formal) de un tip vector se poate face în douã moduri, echivalente:

```
tip nume [ ];          // la fel ca in C si C++
tip [ ] nume;          // specific Java
```

Declararea matricelor (vectori de vectori) poate avea si ea douã forme. Exemplu:

```
int a[ ][ ] ;      // o matrice de întregi
int [ ][ ] b;      // altã matrice de întregi
```

In Java nu este permisã specificarea unor dimensiuni la declararea unor vectori sau matrice, deoarece alocarea de memorie nu se face niciodatã la compilare. Exemplu:

```
int a[100];      // eroare sintacticã !
```

O variabilã vector este automat în Java o variabilã referintã iar memoria trebuie alocatã dinamic pentru orice vector. Alocarea de memorie pentru un vector se face folosind operatorul *new* ,urmat de un nume de tip si de o expresie (cu rezultat întreg) între paranteze drepte; expresia determinã numãrul de componente (nu de octeti !) pe care le poate contine vectorul. Exemple:

```
float x[ ] = new float [10];      // aloca memorie ptr 10 reali
int n=10;
byte[ ][ ] graf = new byte [n][n];
```

Este posibilã si o alocare automatã, atunci când vectorul este initializat la declarare cu un sir de valori. Exemplu:

```
short prime[ ] = {1,2,3,5,7};
```

In lipsa unei initializãri explicite, componentele unui vector sunt initializate automat, cu valori ce depind de tipul lor: zerouri pentru elemente numerice, *null* pentru variabile referintã de orice tip.

Un vector intrinsec cu componente de un anumit tip este considerat ca un obiect de un tip clasã, tip recunoscut de compilator dar care nu este definit explicit în nici un pachet de clase. Numele acestor clase este format din caracterul ‘[‘ urmat de o literã ce depinde de tipul componentelor vectorului: [I pentru int[], [B pentru byte[], [Z pentru boolean[], [C pentru char[], [F pentru float[] s.a.m.d.

Variabila predefinitã cu numele *length* poate fi folositã (ca membru al claselor vector) pentru a obtine dimensiunea alocatã pentru un vector. Exemplu:

```
   // functie de copiere a unui vector
public static void copyVec (  int a [ ] ,int b[ ] )  {
  for (int i=0;i < a.length; i++)              // a.length =dimensiune vector a
      b[i] = a[i];
```

```
}
```

De retinut cã *length* este dimensiunea alocatã si nu dimensiunea efectivã a unui vector, deci numãrul de elemente din vector trebuie transmis ca argument la functii.

In Java, se verificã automat, la executie, încadrarea indicilor între limitele declarate; iesirea din limite produce o exceptie si terminarea programului. Exemplu:

```
int [ ] a= new int [10];
for (int i=1;i<=10;i++)
  a[i]=i;                          // exceptie la  a[10]=10
```

Numerotarea componentelor unui vector este de la zero la (length-1), deci în exemplul anterior se produce exceptia de depãsire a limitelor la valoarea i=10 .

Variabila *length* nu trebuie confundatã cu metoda "length()" din clasa *String*. Functia urmãtoare determinã sirul cel mai lung dintr-un vector de siruri:

```
static String maxLen ( String v[ ]) {
  String max =v[0];
  for (int i=0;i<v.length;i++)
    if ( max.length() < v[i].length() )      // compara lungimile a doua siruri
      max=v[i];                                  // retine adresa sirului cel mai lung
  return max;
 }
```

O matrice este privitã si în Java ca un vector de vectori, iar variabila "length" se poate folosi pentru fiecare linie din matrice.

Deoarece orice matrice este alocatã dinamic, nu existã probleme la transmiterea unei matrice ca argument la o functie. Nu este necesarã transmiterea dimensiunilor matricei ca argumente la functia apelatã. Exemplu:

```
class Matrice {
public static void printmat (int a[ ][ ]) {        // functie de afisare matrice
  for (int i=0;i<a.length;i++) {
    for (j=0; j<a[i].length; j++)
      System.out.print (a[i][j]+ "  ");
      System.out.println ();
    }

public static void main (String [ ] arg) {       // utilizare functie
  int mat[ ][ ]= { {1,2,3}, {4,5,6} };
  printmat (mat);
 }
}
```

O functie poate avea un rezultat de un tip vector.

In Java, ca si în C, transmiterea parametrilor se face prin valoare, adicã se copiazã valorile parametrilor efectivi în parametrii formali corespunzãtori, înainte de executia

functiei. Deci o functie Java nu poate transmite rezultate prin argumente de un tip primitiv, dar poate modifica componentele unui vector primit ca argument. Exemplu:

```
    // creare vector cu divizorii unui întreg
static int divizori (int n, int d[ ]) {
 int k=0;
   for (int i=1;i<n;i++)
     if ( n %i == 0)
        d[k++]=i;                    // pune divizorul i în vectorul d
   return k;                         // numar de divizori
}
    // utilizare functie
  int div[ ]= new int [m];           // aloca memorie ptr vector divizori
  int nd = divizori (m, div);        // completare vector divizori
```

Clasa *Arrays* din pachetul “java.util” reuneste functii pentru operatii uzuale cu vectori având elemente de orice tip (primitiv sau clasã): sortare, cãutare s.a. Exemplu:

```
import java.util.Arrays;
class ExArrays {                                  // utilizare functii din clasa Arrays
  public static void main ( String args[ ) {
     String t[ ] ={ "4","2","6","3","5","1" };    // un vector de siruri
     Arrays.sort (t);  printVec(t);               // ordonare vector t
     int k = Arrays.binarySearch (t, "3");        // cautare in vector ordonat
     float x[ ] = { 3.4, 2.8, 7.1, 5.6 };         // un vector de numere reale
     Arrays.sort (x);  printVec(x);               // ordonare vector x
     float y[ ] = (float[ ]) x.clone();           // y este o copie a vectorului x
     System.out.println ( Arrays.equals (x,y));   // scrie "true"
     System.out.println ( x.equals (y));          // scrie "false"  (metodã a clasei Object)
  }
}
```

Exceptii program în Java

O exceptie program este o situatie anormalã apãrutã la executia unei functii si care poate avea cauze hardware sau software. Exceptiile pot fi privite ca evenimente previzibile, ce pot apare în anumite puncte din program si care afectează continuarea programului, prin abatere de la cursul normal.

Existenta exceptiilor este un mare avantaj al limbajului Java, pentru cã permite semnalarea la executie a unor erori uzuale, prin mesaje clare asupra cauzei si locului unde s-a produs eroarea, evitând efectele imprevizibile ale acestor erori (în C, de ex.).

Exceptiile Java sunt de douã categorii:

- Exceptii care nu necesitã interventia programatorului (numite "Runtime Exceptions"), dar care pot fi interceptate si tratate de către programator. Dacã nu sunt tratate, aceste exceptii produc afisarea unui mesaj referitor la tipul exceptiei si terminarea fortatã a programului. Aceste exceptii corespund unor erori grave care nu permit continuarea executiei si care apar frecvent în programe , cum ar fi: erori de indexare a elementelor unui vector (indice în afara limitelor), utilizarea unei variabile

referintã ce contine *null* pentru referire la date sau la metode publice, împãrtire prin zero, conversie prin "cast" între tipuri incompatibile, s.a.
- Exceptii care trebuie fie tratate, fie "aruncate" mai departe, pentru cã altfel compilatorul marcheazã cu eroare functia în care poate apare o astfel de eroare ("Checked Exceptions": exceptii a cãror tratare este verificatã de compilator).Aceste exceptii corespund unor situatii speciale care trebuie semnalate si tratate, dar nu produc neapãrat terminarea programului. Astfel de exceptii care nu constituie erori sunt : detectare sfârsit de fisier la citire, încercare de extragere element dintr-o stivã sau din altã colectie vidã, încercarea de a deschide un fisier inexistent, s.a.

Urmãtorul program poate produce (cel putin) douã exceptii, dacã este folosit gresit:

```
class Exc {
  public static void main (String arg[ ]) {
    System.out.println ( Integer.parseInt(arg[0]));      // afiseaza primul argument
  }
}
```

O comandã pentru executia programului de forma "java Exc" produce exceptia *ArrayIndexOutOfBoundException*, deoarece nu s-au transmis argumente prin linia de comandã, vectorul "arg" are lungime zero si deci nu existã arg[0] (indice zero).

O linie de comandã de forma "java Exc 1,2" (argumentul arg[0] nu este un sir corect pentru un numãr întreg) produce o exceptie *NumberFormatException*, exceptie generatã în functia "parseInt". Ambele exceptii mentionate erau exceptii "Runtime".

Exceptiile care pot apare la operatii de intrare-iesire (inclusiv la consolã) trebuie fie aruncate, fie tratate pentru cã suntem obligati de cãtre compilatorul Java. Compilatorul "stie" care metode pot genera exceptii si cere ca functiile care apeleazã aceste metode sã arunce mai departe sau sã trateze exceptiile posibile. Exemplu:

```
public static void main (String arg[ ]) throws Exception {
    int ch= System.in.read ( );              // citeste un caracter de la tastatura
    System.out.println ((char) ch);          // afisare caracter citit
}
```

Absenta clauzei *throws* din functia "main" este semnalatã ca eroare de compilator, pentru a obliga programatorul sã ia în considerare exceptia ce poate apare la functia de citire "read", datoritã citirii caracterului EOF (sfârsit de fisier) sau unei erori la citire.

O metodã care apeleazã o metodã ce aruncã exceptii verificate trebuie fie sã semnaleze mai departe posibilitatea aparitiei acestei exceptii (prin clauza *throws*), fie sã trateze exceptia printr-un bloc *try-catch* care sã includã apelul metodei.

Cel mai simplu este ca exceptiile ce pot apare si care nu pot fi ignorate sã fie aruncate mai departe, ceea ce are ca efect oprirea executiei programului la producerea exceptiei si afisarea unui mesaj explicativ. Asa s-a procedat în exemplul anterior.

Tratarea exceptiilor necesitã folosirea unui bloc *try-catch* pentru delimitarea sectiunii de cod pentru care exceptiile posibile sunt redirectate cãtre secvente scrise de utilizator pentru tratarea exceptiilor proiduse. Exemplu :

```
public static void main (String arg[]) {
    Object ref=null;                          // un pointer nul
    try { int h = ref.hashCode();  }      // aici poate apare o exceptie daca ref==null
    catch (NullPointerException e) {
      System.out.println ( e.getMessage());
    }
}
```

Este preferabilã aruncarea unei exceptii fatã de tratarea prin ignorarea exceptiei, care împiedicã aparitia unui mesaj de avertizare la producerea exceptiei. Exemplu:

```
public static void main (String arg[]) {
    Object ref=null;
    try {int h = ref.hashCode(); }              // exceptie daca ref==null
    catch (NullPointerException e) { }        // interzice afisare mesaj (nerecomandat!)
}
```

O functie poate contine unul sau mai multe blocuri *try* , iar un bloc *try* poate contine una sau mai multe instructiuni, în care pot apare exceptii. Un bloc *try* se terminã cu una sau mai multe clauze *catch* pentru diferite tipuri de exceptii care pot apare în bloc.

In exemplul urmãtor se foloseste un singur bloc *try* pentru diferite exceptii posibile:

```
try {  f= new RandomAccessFile(arg[0],"r");        // deschide fisier pentru citire
       while ( true)
         System.out.println ( f.readInt());          // citeste si scrie un numar
} catch  (IOException e) { }                         // ignora orice exceptie de intrare-iesire
```

Varianta urmãtoare foloseste douã blocuri try pentru a trata separat exceptiile:

```
try {
    f= new RandomAccessFile(arg[0],"r");       // deschide fisier pentru citire
} catch  (IOException e) {                          //  exceptie de fisier negasit
   System.out.println("Eroare la citire fisier");
 }
try {
   while ( true)
       System.out.println ( f.readInt());          // citeste si scrie un numar
} catch  (IOException e) { }                         // exceptie de sfarsit de fisier la citire
```

Varianta urmãtoare foloseste un singur bloc *try*, dar separã fiecare tip de exceptie:

```
try {
    f= new RandomAccessFile("numer","r");
    while ( true)
      System.out.println ( f.readInt());
  }
  catch (FileNotFoundException e) {System.out.println ("Fisier negasit"); }
```

```
catch  (EOFException e) { }
catch (IOException e) { System.out.println("Eroare la citire fisier");
}
```

Este posibilã si aruncarea unor exceptii putin probabile (exceptia de citire, de ex.) combinatã cu tratarea altor exceptii (de exemplu exceptia de sfârsit de fisier).

Producerea unor exceptii poate fi prevenitã prin verificări efectuate de programator, ca în exemplele urmãtoare:

```
void fun (Object obj) {
  if (obj==null)  { System.out.println ("Null Reference"); System.exit(-1); }
  . . .
}

public static void main (String arg[ ]) {
  if (arg.length == 0 ) { System.out.println ("No Argument"); System.exit(-1); }
  System.out.println ( arg[0]);      // afiseaza primul argument
}
```

Uneori este preferabilã verificarea prin program (ca în cazul unor conversii de tip nepermise), dar alteori este preferabilã tratarea exceptiei (ca în cazul detectării existentei unui fisier înainte de a fi deschis, sau a utilizării unor variabile referintã ce pot fi nule).

Utilizarea masinii virtuale Java

Codul intermediar generat de compilatoarele Java (numit “bytecode”) este interpretat sau executat într-o masinã virtualã Java.

Interpretorul încarcã automat sau la cerere clasele necesare pentru executia functiei “main”(din fisiere de tip “class” sau de tip “jar” locale sau aduse prin retea de la alte calculatoare). Acest mod de lucru face posibilã utilizarea de clase cu nume necunoscut la executie sau înlocuirea unor clase, fãrã interventie în codul sursã (cu conditia ca aceste clase sã respecte anumite interfete).

Impreunã cu codul clasei (metodele clasei în format compilat) se mai încarcã si informatii despre clasã, numite si metadate: tipul clasei (sau interfetei), numele superclasei, numele variabilelor din clasã, numele si tipul metodelor clasei, formele de constructori pentru obiectele clasei s.a. Aceste metadate permit obtinerea de informatii despre clase la executie, instantierea de clase cunoscute numai la executie, apeluri de metode determinate la executie si alte operatii imposibile pentru un limbaj compilat.

Mai nou, se pot adãuga metadate utilizabile de cãtre diverse programe înainte de executie sau chiar în cursul executiei (“annotations” =adnotări la metadate).

In plus, utilizatorii au posibilitatea de a modifica anumite aspecte ale comportării masinii virtuale Java.

Semnalarea erorilor prin exceptii si informatiile furnizate despre exceptii se datoreazã tot executiei programelor Java sub controlul masinii virtuale.

Codul intermediar Java este “asistat” (supravegheat) la executie de către masina virtuală, ceea ce a dat nastere expresiei de cod controlat (“managed code”) în .NET.

Aceste facilitãti, alãturi de independenta fatã de procesorul pe care se executã, explicã de ce limbajele noi orientate pe obiecte sunt si interpretate (Java si C#).

Biblioteci de clase Java

Limbajul Java este însotit de mai multe biblioteci de clase predefinite, unele direct utilizabile, altele folosite ca bazã pentru definirea altor clase. Clasele Java sunt foarte diferite ca numãr de functii (metode) si ca dimensiune: de la câteva metode la câteva zeci de metode, cu mii de linii sursã. Prin "clase" întelegem aici clasele cu metode statice, clasele instantiabile, clasele abstracte si interfete Java.

Clasele sunt grupate în pachete de clase (biblioteci) în functie de aplicatiile cãrora le sunt destinate, sau de problemele rezolvate pentru mai multe categorii de aplicatii. Initial toate pachetele erau grupate în directorul "java", dar ulterior a fost creat un alt director important "javax" (cu "extensii" ale limbajului Java). Numãrul de pachete si continutul acestora se modificã la fiecare nouã versiune JDK (SDK), prin extindere.

Directorul (pachetul) ”org” contine clase provenite din alte surse decât firma “Sun Microsystem”, dar integrate în bibliotecile standard Java (parsere XML, de ex.).

Dezvoltarea anumitor categorii de aplicatii se poate face mult mai rapid în Java decât în C sau C++, folosind clasele JDK pentru a crea interfata graficã a aplicatiei, pentru lucrul cu colectii de date, pentru prelucrãri de texte, pentru operatii de intrare-iesire, pentru comunicarea cu alte calculatoare din retea etc.

Clasele unor pachete sunt concepute pentru a fi folosite împreunã si pentru a crea noi clase prin derivare; ele creeazã un cadru ("Framework") pentru dezvoltarea de aplicatii cu o structurã unitarã. Astfel de familii de clase sunt clasele colectie (grupate în pachetul "java.util"), clasele de intrare-iesire (pachetele "java.io" si "java.nio"), clasele pentru crearea de interfete grafice ("javax.swing"), clase pentru aplicatii cu baze de date ("java.sql"), pentru aplicatii de retea ("java.net"), pentru prelucrarea de fisiere XML ("javax.xml" si altele), clase pentru prelucrarea documentaiei extrase din surse Java ("com.sun.javadoc") etc.

Clasele Java sunt bine documentate (fisiere descriptive HTML si fisiere sursã), iar utilizarea lor este ilustratã printr-un numãr mare de exemple în "Java Tutorial" si prin programe demonstrative (subdirectorul "demo").

Un mediu integrat Java permite consultarea documentatiei claselor chiar în cursul editării, ceea ce faciliteazã utilizarea acestui numãr imens de clase si de metode.

2. Introducere în programarea orientatã pe obiecte

Clase si obiecte

Programarea cu obiecte (POO) reprezintã un alt mod de abordare a programãrii decât programarea proceduralã (în limbaje ca C sau Pascal), cu avantaje în dezvoltarea programelor mari.

Un program procedural (scris în C de ex.) este o colectie de functii, iar datele prelucrate se transmit între functii prin argumente (sau prin variabile externe). In programarea proceduralã sarcina programatorului este de a specifica actiuni de prelucrare, sub formã de proceduri (functii, subprograme).

Exemplul urmãtor este o functie C de copiere a unui fisier, octet cu octet:

```
 // copiere fisier in C
void filecopy ( char * src, char * dst) {
   char ch;
   FILE * in =fopen(src,"r");        // deschide fisier sursa
   FILE * out =fopen(dst,"w");     // deschide fisier destinatie
   while ( (ch=fgetc (in)) != -1)    // citeste un caracter
     fputc (ch, out);               // scrie un caracter
   fclose(out); fclose(in);
}
```

In acest exemplu se apeleazã functiile fopen, fgetc, fputc, fclose care primesc ca date variabilele “in”, “out” si “ch”.

Un program orientat pe obiecte este o colectie de obiecte care interactioneazã prin apeluri de functii specifice fiecãrui tip de obiect. In programarea orientatã pe obiecte programatorul creeazã obiectele necesare aplicatiei si apeleazã metode ale acestor obiecte pentru actiuni de prelucrare a datelor continute în obiectele aplicatiei. Un obiect contine în principal date.

Exemplul urmãtor este o functie Java de copiere a unui fisier, octet cu octet:

```
public static void filecopy (String src, String dst) throws IOException {
    FileReader in = new FileReader (src);       // un obiect
    FileWriter out = new FileWriter (dst);        // alt obiect
    int ch;
    while ( (ch= in.read()) != -1)           // cere obiectului “in” operatia “read”
      out.write(ch);                          // cere obiectului “out” operatia “write”
    in.close(); out.close();                  // cere obiectelor operatia “close”
  }
```

In acest exemplu se folosesc douã obiecte: un obiect de tip *FileReader* (prin variabila “in”) si un obiect de tip *FileWriter* (prin variabila “out”); prin metoda “read” se cere obiectului “in” sã citeascã si sã furnizeze un caracter, iar prin metoda “write” se cere obiectului “out” sã scrie în fisier caracterul primit ca argument.

Pentru obiectele “in” si “out” se pot apela si alte metode, din clasele respective.

Definitia unei clase poate fi privitã ca o extindere a definirii unui tip structurã din C, care contine si functii pentru operatii cu datelor continute în clasã. Aceste functii se numesc si metode ale clasei. In Java si în C++ functiile (metodele) sunt subordonate claselor; o metodã poate fi aplicatã numai obiectelor din clasa care contine si metoda.

Un obiect corespunde unei variabile structurã din C. In C++ o variabilã de un tip clasã este un nume pentru un obiect, dar în Java o variabilã de un tip clasã contine un pointer cãtre un obiect (corespunde unei variabile referintã din C++). De aceea, mai corectã este exprimarea ‘se apeleazã metoda “read” pentru obiectul adresat prin variabila “in”’. Este posibil ca mai multe variabile de un tip clasã sã continã adresa aceluiasi obiect, desi în practicã fiecare variabilã Java se referã la un obiect separat.

Obiectele Java sunt create dinamic, folosind de obicei operatorul *new*, care are ca rezultat o referintã la obiectul alocat si initializat printr-o functie constructor, apelatã implicit de operatorul *new*.

O clasã corespunde unei notiuni abstracte cum ar fi “orice fisier disc”, iar un obiect este un caz concret (o realizare a conceptului sau o instantiere a clasei). Un obiect de tip *FileReader* corespunde unui anumit fisier, cu nume dat la construirea obiectului.

In general, obiecte diferite contin date diferite, dar toate obiectele suportã aceleasi operatii, realizate prin metodele clasei din care fac parte.

Relativ la exemplul Java, trebuie spus cã utilizarea unei functii statice de copiere nu este în spiritul POO. Functiile statice Java corespund functiilor C si pot fi folosite fãrã ca sã existe obiecte (ele fac parte totusi din anumite clase).

Mai aproape de stilul propriu POO, ar trebui definitã o clasã copiator de fisiere, având si o metodã (nestaticã) “filecopy”, cu sau fãrã argumente. Exemplu:

```
public class FileCopier {
     // datele clasei
  private  FileReader in;         // sursa datelor
  private  FileWriter out;        // destinatia datelor
      // constructor
  public FileCopier (String src, String dst) throws IOException {
    in = new FileReader (src);
    out = new FileWriter (dst);
  }
     // o metoda de copiere
  public void filecopy () throws IOException {
    int c;
    while ( (c= in.read()) != -1)
      out.write(c);
    in.close(); out.close();
  }
}
   // clasa pentru verificarea clasei FileCopier
class UseFileCopier {
  public static void main (String arg[]) throws IOException {
    FileCopier fc = new FileCopier (arg[0], arg[1]);          // creare obiect “fc”
```

```
        fc.filecopy();       // cere obiectului “fc” operatia “filecopy”
    }
}
```

Clasa “FileCopier”, definitã anterior, trebuie privitã doar ca un prim exemplu, ce trebuia sã fie foarte simplu, si nu ca un model de clasã realã Java.

In exemplele anterioare si în cele ce vor urma se poate observa mutarea accentului de pe actiuni (functii) pe obiecte (date) în programarea orientatã pe obiecte. Numele de clase sunt substantive, uneori derivate din verbul ce defineste principala actiune asociatã obiectelor respective. In Java existã obiecte comparator (de un subtip al tipului *Comparator*) folosite în compararea altor obiecte, clasa *Enumerator* folositã la enumerarea elementelor unei colectii, clasa *StringTokenizer* folositã la extragerea cuvintelor dintr-un sir s.a.

Clasele sunt module de program reutilizabile

Definirea si utilizarea de module functionale permite stãpânirea complexitãtii programelor mari si reutilizarea de module prin crearea de biblioteci.

In limbajul C un modul de program este o functie, dar în Java si C++ un modul este o clasã, care reuneste în general mai multe functii în jurul unor date.

Utilizarea de clase ca module componente ale programelor are o serie de avanaje fatã de utilizarea de functii independente:

- Metodele unei clase necesitã mai putine argumente, iar aceste argumente nu sunt modificate în functie; efectul unei metode este fie de a face accesibile date din clasã, fie de a modifica variabile din clasã pe baza argumentelor primite. Variabilele unei clase sunt implicit accesibile metodelor clasei si nu mai trebuie transmise explicit, prin argumente (ca niste variabile externe metodelor, dar interne clasei).
- Solutii mai simple pentru functii al cãror efect depinde de stare (de context), cum ar fi de apeluri anterioare ale aceleeasi functii sau ale altor functii pregãtitoare.
- O clasã poate încapsula algoritmi de complexitate ridicatã, realizati prin colaborarea mai multor functii, unele interne clasei; astfel de algoritmi fie nu sunt disponibili în C, fie sunt disponibili prin biblioteci de functii destul de greu de utilizat. Exemple sunt algoritmi pentru lucrul cu expresii regulate, pentru arhivare-dezarhivare, pentru operatii cu anumite structuri de date (arbori binari cu auto-echilibrare), s.a.
- Se poate realiza un cuplaj mai slab între module, în sensul cã modificarea anumitor module nu va afecta restul programului. Aceastã decuplare sau separare între module se poate realiza prin mai multe metode, printre care folosirea de interfete Java, în spatele cãrora pot sta clase cu implementãri diferite dar cu acelasi mod de utilizare.

Pentru a concretiza aceste afirmatii vom prezenta comparativ solutiile C si Java pentru câteva probleme de programare.

Primul exemplu se referã la utilizarea structurii de tip stivã (“stack”) în aplicatii. O stivã poate fi realizatã fie printr-un vector, fie printr-o listã înlãntuitã, dar operatiile cu stiva sunt aceleasi, indiferent de implementare: pune date pe stivã, scoate datele din vârful stivei si test de stivã goalã.

In limbajul C se pot defini functiile pentru operatii cu stiva astfel ca utilizarea lor sã nu depindã de implementarea stivei, prin folosirea unui pointer la o structurã:

```
void initSt ( Stiva *  sp);    // initializare stiva
int emptySt (Stiva * s);       // test stiva goala
int push (Stiva * sp, T x);  // pune in stiva un element de tip T
T pop (Stiva * sp );       // scoate din stiva un element de tip T
```

Definitia tipului “Stiva” si definitiile functiilor depind de implementare. Exemple:

```
  // stiva ca lista inlantuita
typedef struct snod {
       T val;
       struct snod * leg;
       } nod, * Stiva ;
   // stiva vector
typedef struct {
     T * st ;          // adresa vector (alocat dinamic)
     int sp;           // indice varf stiva
     } Stiva;
```

Un exemplu de utilizare a unei stive:

```
#include "stiva.h"
#define T int
void main () {
 int x; Stiva s ;
 initSt (&s);                       // initializare stiva
 for (x=1; x<10; x++)               // genereaza date ptr continut stiva
    push (&s,x);                    // pune x pe stiva
 while ( ! emptySt (&s) )           // cat timp stiva contine ceva
   printf("%d \n", pop (&s) ) ;   // scoate din stiva si afiseaza
}
```

Modificarea tipului de stivã necesitã un alt fisier “stiva.h” si o altã bibliotecã de functii (push, pop), care sã fie folositã împreunã cu programul de utilizare a stivei.

In Java acelasi program de exersare a operatiilor cu stiva aratã astfel:

```
public static void main (String arg[ ]) {
   Stack s = new Stack();
   for (int x=1; x<10; x++)
      s.push ( new Integer(x));       //  s.push(x)  in Java 5
    while ( ! s.empty())
       System.out.println ( s.pop());
}
```

Modificarea implementãrii stivei, prin definirea unei alte clase “Stack” nu necesitã modificãri în functia anterioarã, ci doar punerea noii clase în cãile de cãutare ale compilatorului si interpretorului Java. In plus, modificarea tipului datelor puse în stivã necesitã modificãri mai mici în Java decât în C.

Un al doilea exemplu este cel al extragerii de cuvinte succesive dintr-un sir de caractere ce poate contine mai multe cuvinte, separate prin anumite caractere date. Problema este aceea cã dupã fiecare cuvânt extras se modificã adresa curentã în sirul analizat, deci starea sau contextul în care se executã functia ce da urmãtorul cuvânt.

In limbajul C se pot întâlni mai multe solutii ale acestei probleme în diferite functii de bibliotecã:

Functia “strtok” se foloseste relativ simplu, dar pretul plãtit este modificarea sirului analizat si imposibilitatea de a analiza în paralel mai multe siruri (pentru cã adresa curentã în sirul analizat este o variabilã staticã internã a functiei). In plus, primul apel diferã de urmãtoarele apeluri ale functiei. Exemplu:

```
cuv=strtok(str, sep);       // primul cuvânt din “str”, sep= sir de separatori
while (cuv !=NULL) {        // daca s-a gasit un cuvant
  puts(cuv);                // afisare cuvant
  cuv=strtok(0,sep);        // urmatorul cuvant din “str”
}
```

Functia “strtod” extrage urmãtorul numãr dintr-un sir si furnizeazã adresa imediat urmãtoare numãrului extras. Exemplu de utilizare:

```
char * p = str; double d;       // str = sir analizat
do {
  d= strtod (p,&p);      // cauta de la adresa p si pune tot in p adresa urmatoare
  printf ("%lf\n", d);
} while (d != 0);          // d=0 cand nu mi exista un numar corect
```

In Java existã clasa de bibliotecã *StringTokenizer*, folositã dupã cum urmeazã:

```
String sep = new String (" ,.;\n\t");          // lista separatori de cuvinte
StringTokenizer st = new StringTokenizer (sir,delim);   // “sir” = sir analizat
 while (st.hasMoreTokens()) {                  // daca mai sunt cuvinte in sirul analizat
    String token = st.nextToken();             // extrage urmatorul cuvint din linie
    System.out.println (token);                // afisare cuvint
 }
```

La crearea unui obiect *StringTokenizer* se specificã sirul analizat, astfel cã se pot analiza în paralel mai multe siruri, pentru fiecare folosind un alt obiect. Metodele “nextToken” si “hasMoreTokens” folosesc în comun o variabilã a clasei care contine pozitia curentã în sirul analizat (initializatã cu adresa sirului, la construirea obiectului).

In prelucrarea fisierelor apar situatii când executia cu succes a unei functii depinde de folosirea anterioarã a altor functii (cu anumite argumente); de exemplu pentru a putea scrie într-un fisier, acesta trebuie anterior deschis pentru creare sau pentru adãugare (extindere fisier existent). O situatie asemãnãtoare apare la utilizarea unor functii care compun o interfatã graficã si care trebuie folosite într-o anumitã ordine.

Astfel de conditionãri reciproce nu se pot verifica automat în C, fiind vorba de functii independente. In Java operatia de deschidere fisier si operatia de scriere sunt

metode dintr-o aceeasi clasã si se poate verifica printr-o variabilã a clasei succesiunea corectã de folosire a metodelor.

Functionalitatea unei clase poate fi reutilizatã în alte clase fie prin derivare, fie prin agregare (compunere). In acest fel, operatiile necesare într-o clasã sunt fie mostenite de la o altã clasã, fie delegate spre executie metodelor unei alte clase. De exemplu, extinderea automatã a unui vector, necesarã dupã anumite operatii de adãugare la vector, este refolositã si într-o clasã stivã vector, fie prin definirea clasei stivã ca o clasã derivatã din vector, fie prin folosirea unei variabile *Vector* în clasa stivã.

In POO adaptarea unei clase la cerinte specifice unor aplicatii nu se face prin interventie în codul clasei ci prin derivare sau prin delegare, tehnici specifice POO.

O interfatã contine una sau mai multe operatii (metode abstracte) cu rol bine definit, dar a cãror implementare nu poate fi precizatã. Cel mai simplu exemplu din Java este interfata *Comparator*, cu o singurã metodã “compare”, pentru compararea a douã obiecte (dupã modelul comparatiei de siruri din C). Pentru fiecare tip de obiecte (comparabile) va exista o altã definitie a functiei “compare”.

O interfatã cu o singurã metodã corespunde unui pointer la o functie din limbajul C, dar interfetele cu mai multe metode creeazã posibilitãti inexistente în C. O interfatã poate fi implementatã de mai multe clase, toate cu acelasi rol dar cu mod de lucru diferit. Interfata *Collection* defineste câteva operatii ce trebuie sã existe în orice colectie de obiecte, indiferent de structura colectiei: adãugare obiect la colectie s.a.

O clasã creeazã un spatiu de nume pentru metodele clasei: pot exista metode cu acelasi nume (si aceleasi argumente) în clase diferite. Pachetul de clase (“package”) este o altã unitate Java care creeazã un spatiu de nume pentru clasele continute în el.

O notiune proprie programãrii cu obiecte este notiunea de componentã software. Ideea este de a obtine rapid un prototip al aplicatiei fãrã a scrie cod sau cu un minim de programare, prin asamblarea de componente prefabricate (pentru interfata graficã).

O componentã poate contine una sau mai multe clase si poate fi reutilizatã si adaptatã fãrã interventie în codul sursã al componentei (care nici nu este disponibil).

O componentã JavaBeans poate fi (re)utilizatã fãrã a scrie cod, prin generarea automatã a operatiilor de instantiere, de modificare a proprietãtilor si de conectare cu alte clase (prin apeluri de metode sau prin evenimente), în urma unor comenzi date de utilizator unui mediu vizual de dezvoltare a aplicatiilor. O componentã este de obicei o clasã care respectã anumite conditii.

Clasele creeazã noi tipuri de date

In limbajul C, prin definirea de tipuri structurã, se pot defini noi tipuri de date ca grupuri de variabile de alte tipuri predefinite. De exemplu, vom defini o structurã cu douã variabile (parte realã, parte imaginarã) pentru un numãr complex:

```
typedef struct { float re; float im; } Complex ;
```

Operatii cu variabile de acest nou tip se definesc prin functii, cu argumente de tip “Complex” (sau “Complex*” dacã functia modificã parametrul primit).

Multe din clasele de bibliotecã Java pot fi privite ca având drept scop extinderea limbajului cu noi tipuri de date, utile în mai multe aplicatii. Exemple: BigInteger, BigDecimal, String, Date etc.

Cel mai folosit tip de date definit printr-o clasã este tipul *String*; un obiect de tip *String* contine ca date un vector de caractere si lungimea sa si suportã un numãr mare de metode ce corespund unor operatii uzuale cu siruri (realizate prin functii de bibliotecã sau prin functii definite de utilizatori, în limbajul C).

Noile tipuri de date se pot folosi în declaratii de variabile, de functii, de argumente de functii. Exemplu de functie staticã care foloseste metodele “indexOf”, “length” si “substring” din clasa *String* :

```
   // inlocuire repetata in s a lui s1 prin s2
public static String replaceAll (String s, String s1, String s2) {
  int p;
  p = s.indexOf(s1);
  while ( p >=0 ) {
     s = s.substring(0,p)+ s2 + s.substring(p+s1.length());
     p=s.indexOf(s1, p+s2.length());
  }
  return s;
}
```

Utilizatorii îsi pot defini propriile clase, pentru tipuri de date necesare aplicatiilor; de exemplu, putem defini o clasã “BoolMatrix” pentru o matrice cu elemente de tip *boolean*. In Java orice clasã este automat derivatã dintr-o clasã generică *Object* si, ca urmare, trebuie sã redefineascã anumite metode mostenite: “toString”, “equals” s.a.

Exemplu de clasã minimalã pentru o matrice de biti:

```
   // matrice cu elemente de tip boolean
public class BoolMatrix {
  private boolean a[ ][ ];       // o matrice patratica
  private int n;         // nr de linii si coloane
   // constructor de obiecte
  public BoolMatrix (int n) {
    this.n=n;
    a= new boolean[n][n];       // aloca memorie ptr matrice
  }
   // modifica valoare element
  public void setElement (int i,int j, boolean b) {
     a[i][j]=b;
  }
   // citire valoare element
  public boolean getElement (int i,int j) {
     return a[i][j];
  }
   // sir cu elementele din matrice
  public String toString () {
     String s="";
```

```
    for (int i=0;i<n;i++) {
      for (int j=0;j<n;j++)
        s=s + ( a[i][j]==true ? "1 ": "0 ") ;
      s=s+"\n";
    }
    return s;
  }
    // exemplu de utilizare
  public static void main (String arg[]) {
    BoolMatrix mat = new BoolMatrix(4);
    for (int i=0;i<4;i++)
      mat.setElement (i,i,true);
    System.out.println ( mat.toString());      // sau  System.out.println (mat);
  }
}
```

Variabilele dintr-o clasã sunt declarate de obicei cu atributul *private*, ceea ce le face inaccesibile pentru metode din alte clase. Se mai spune cã datele sunt ascunse sau sunt încapsulate în fiecare obiect. Metodele clasei sunt de obicei publice pentru a putea fi apelate din alte clase.

Deoarece datele dintr-un obiect (variabile private) nu sunt direct accesibile din afara clasei si pot fi modificate numai prin intermediul metodelor clasei, utilizarea tipurilor de date definite prin clase este mai sigurã decât a celor definite prin structuri. De exemplu, orice modificare a vectorului de caractere dintr-un obiect *StringBuffer* este însotitã de modificarea lungimii sirului (în metodele care pot modifica lungimea sirului), dar lungimea nu poate fi modificatã direct de cãtre functii din alte clase (si nici continutul vectorului de caractere).

In Java nu se pot supradefini operatori, deci operatiile cu noile tipuri de date se pot exprima numai prin metode asociate obiectelor (metode nestatice).

Tipurile de date definite prin clase pot forma ierarhii de tipuri compatibile (care se pot înlocui prin atribuire sau la transmitere de argumente).

Clasele permit programarea generică

Programarea generică ne permite sã avem o singurã clasã pentru un vector (sau pentru o listã), indiferent de tipul datelor care vor fi memorate în vector (în listã).
Tot programarea generică ne permite sã folosim o singurã functie (metodã) pentru a parcurge elementele oricãrei colectii (indiferent de structura ei fizicã) sau pentru a ordona orice listã abstractã (o colectie care suportã acces direct prin indice la orice element din colectie).

Genericitatea poate fi realizatã în POO în douã moduri: prin crearea unor ierarhii de clase (de tipuri) sau prin clase sablon, cu tipuri parametrizate (“templates”).

Dintr-o clasã se pot deriva alte clase, pe oricâte niveluri. O clasã derivatã (numitã si subclasã) preia prin mostenire toate datele clasei de bazã (numitã si superclasã) si metodele publice. La membri mosteniti subclasa poate adãuga alte date sau metode.

Pe lângã reutilizarea metodelor din superclasã în subcasã, derivarea creează tipuri compatibile si ierarhii de tipuri. Tipul unei clase derivate este subtip al tipului clasei

din care derivã, asa cum tipul *int* poate fi considerat ca un subtip al tipului *long*, iar tipul *float* ca un subtip al tipului *double*.

La fel cum un argument formal de tip *double* poate fi înlocuit cu un argument efectiv de tip *int*, tot asa un argument formal de un tip clasã B poate fi înlocuit cu un argument efectiv de un tip clasã D; clasa D fiind derivatã din clasa B. In felul acesta se pot scrie functii generice, cu argumente de un tip general si utilizabile cu o multitudine de tipuri de argumente (asa cum functia "sqrt" se poate apela cu argument de orice tip numeric din C).

In Java toate clasele predefinite sau care urmeazã a fi definite de utilizatori sunt implicit derivate dintr-o clasã generică *Object*, care este superclasa directã sau indirectã a oricãrei clase.

O colectie de variabile de tip *Object* este o colectie generică, pentru cã acestor variabile li se pot atribui variabile de orice alt tip clasã (care contin adresele unor obiecte). Toate clasele colectie Java sunt colectii generice. Exemplu de utilizare a unui obiect de tip *Vector* pentru memorarea unor siruri (obiecte de tip *String*):

```
Vector v = new Vector();                  // creare obiect vector (extensibil)
String a[] = {"unu","doi","trei"};     // un vector intrinsec (ca în C)
for (int k=0; k<a.length; k++)       // parcurge tot vectorul a
   v.add ( a[k]);                          // adauga sirul a[k] la vector
 System.out.println (v);                 // afisare continut vector
 String s = (String) v.elementAt (0);       // s va contine un pointer la sirul "unu"
```

Conversia în sus de la subtipul *String* la supertipul *Object* se face automat (argumentul metodei "add" este de tip *Object*), dar conversia în jos (de la *Object* la *String*, pentru rezultatul metodei "elementAt") trebuie cerutã în mod explicit prin operatorul de conversie (ca si în C).

Genericitatea în POO este sustinutã si de existenta metodelor polimorfice, precum si a iteratorilor, ca mecanism de parcurgere a unei colectii abstracte.

O metodã polimorficã este o metodã care se foloseste la fel pentru diferite tipuri de obiecte, desi implementarea ei este diferitã de la o clasã la alta. Majoritatea metodelor Java sunt (implicit) polimorfice : equals, toString, add, compareTo, etc.

Un iterator este un obiect cu metodele (polimorfice) "next" si "hasNext", prin care putem accesa succesiv elementele unei colectii. Exemplul urmãtor este o metodã care afiseazã continutul oricãrei colectii, indiferent de tipul colectiei si de tipul obiectelor continute, nivel de generalizare greu de atins într-un limbaj procedural:

```
public static void print (Collection c) {
   Iterator it = c.iterator();              // creare obiect iterator ptr colectia c
   while ( it.hasNext())              // repeta cat timp mai sunt elemente in colectie
      System.out.print (e.next()+"  ");    // scrie valoare element curent si avans
  System.out.println();
}
```

Clasele creeazã un model pentru universul aplicatiei

Un program destinat unei aplicatii trebuie sã transforme notiunile si actiunile specifice aplicatiei în constructii specifice limbajului de programare folosit (functii, variabile, argumente de functii, etc.).

Evolutia limbajelor de programare poate fi privitã si ca un progres al abstractizãrii, în sensul îndepãrtãrii progresive de masina fizicã prin introducerea de notiuni tot mai abstracte. Fiecare limbaj de programare oferã programatorilor o masinã virtualã (sau abstractã) diferitã de masina concretã pe care se vor executa programele lor.

Programarea orientatã pe obiecte permite definirea de clase si obiecte ce corespund direct obiectelor din universul aplicatiei si modelarea relatiilor statice si dinamice dintre aceste obiecte. Identificarea obiectelor si actiunilor specifice unei aplicatii se face în faza de analizã orientatã obiect a problemei de rezolvat si implicã o abordare diferitã de cea anterioarã.

Un program Java poate fi privit ca o descriere a unor obiecte si a interactiunilor dintre aceste obiecte. Intr-un program Java nu existã altceva decât obiecte si clase.

O analizã orientatã pe obiecte poate începe cu o descriere în limbaj natural a ceea ce trebuie sã facã programul; substantivele din acest text corespund în general unor obiecte (clase), iar verbele din text corespund unor metode. O astfel de abordare este potrivitã pentru aplicatii grafice, pentru jocuri, pentru unele aplicatii economice s.a.

Multe aplicatii folosesc o interfatã graficã cu utilizatorii (operatorii) aplicatiei; obiectele vizuale afisate pe ecran si care pot fi selectate sau actionate de operator (ferestre, butoane, meniuri, casete cu text, s.a.) corespund direct unor obiecte din programele Java.

Intr-o aplicatie bancarã vor exista clase si obiecte de genul “Account” (cont bancar) si “Customer” (client al bãncii). Un obiect “Customer” va contine date de identificare ale clientului si metode pentru obtinerea sau modificarea acestor date. Un obiect de tip “Account” va contine suma din cont (si alte date asupra operatiilor cu acel cont), precum si metode pentru depunerea de bani în cont, pentru retragerea de bani din cont si pentru vizualizarea sumei de bani din cont.

Obiectele de tipul “Account” sau “Customer” se numesc si obiecte din domeniul aplicatiei (“domain objects”). Aplicatiile mai pot contine obiecte ajutãtoare (“helper”) sau obiecte din clase predefinite pentru operatii cu anumite tipuri de date, cu colectii de obiecte, cu baze de date, cu conexiuni între calculatoare s.a.

Consideratii de proiectare în vederea schimbãrii (“design for change”) pot introduce clase si obiecte suplimentare, de obicei parte a unor scheme de proiectare (“design patterns”) destinate a reduce cuplajul dintre diferite pãrti ale aplicatiei.

3. Utilizarea de clase si obiecte

Clase fãrã obiecte. Metode statice.

Metodele Java sunt de douã categorii:
- Metode aplicabile obiectelor (“Object Methods”)
- Metode statice, utilizabile independent de obiecte (“Class Methods”)

O metodã staticã Java corespunde unei functii din limbajul C, dar poate fi folositã numai precedatã de numele clasei. In felul acesta putem avea functii cu acelasi nume în clase diferite si se reduce posibilitatea unui conflict de nume.

Câteva clase Java nu sunt altceva decât grupãri de functii statice relativ independente. Aceste clase nu sunt instantiabile, deci nu pot genera obiecte. Metodele statice sunt în general si publice, pentru a putea fi apelate din orice altã clasã. Exemple de utilizare:

```
    // afisare radical dintr-un numar primit in linia de comanda
class Sqrt  {
  public static void main (String args[ ]) {
    int x = Integer.parseInt (args[0]);              // functie statica din clasa Integer
    double r = Math.sqrt (x);                        // functie statica din clasa Math
    System.out.println (r);
  }
}
```

In Java se citesc din fisiere sau se preiau din linia de comandã siruri de caractere, iar analiza lor si conversia în format intern pentru numere se face explicit de cãtre programator. Din versiunea 1.5 au fost introduse în Java metode similare functiilor "scanf" si "printf" din C. Exemple:

```
    System.out.printf ("%s %5d\n", name, total);       // afisare cu format
    Scanner s=Scanner.create(System.in);               // ptr citire cu format
    String param= s.next();              // metoda ptr citire sir
    int value=s.nextInt();               // metoda ptr citire numar intreg
```

Pentru functiile matematice nu existã altã posibilitate de definire decât ca metode statice, deoarece ele primesc un parametru de un tip primitiv (*double*). Pentru metodele cu un operand de un tip clasã avem de ales între o functie ne-staticã si o functie staticã. Metodele statice au un parametru în plus fatã de metodele nestatice cu acelasi efect. Exemple:

```
    // o metoda statica pentru conversie numar din Integer in int
public static int getInt (Integer a) {
    return a.intValue ();                        // apel metodã nestatica
}
  // exemplu de utilizare pentru conversie de la String la int
Integer a = new Integer (str);
```

```
int x = getint (a);                    // sau  x = new Integer(str).intValue();
```

O metodã staticã se poate referi numai la variabile statice din clase (variabile definite în afara functiilor si cu atributul *static*). Exemplu:

```
class A {
  static String msg = "O.K.";                    //  "static" este necesar aici
  public static void main  (String arg[ ]) {  System.out.println (msg);  }
}
```

Clase instantiabile. Metode aplicabile obiectelor

Specific programãrii orientate pe obiecte este utilizarea de clase care contin si date si care pot genera obiecte. O astfel de clasã poate fi privitã ca un sablon pentru crearea de obiecte care au în comun aceleasi operatii (metode) dar contin date diferite. Un obiect este un caz particular concret al unei clase, deci o "instantiere" a unei clase.

Intr-un program Java se creeazã obiecte si se apeleazã metode ale acestor obiecte. In exemplul urmãtor se afiseazã numãrul fisierelor dintr-un director, al cãrui nume este primit în linia de comandã, folosind un obiect de tipul *File*:

```
import java.io.File;
class FileDir  {
  public static void main (String arg[ ]) throws IOException {
     File d = new File (arg[0]);           //  creare obiect de tip File
     String[ ] files ;                     // un vector de siruri
     if ( d.isDirectory()) {               //   apel metoda ptr obiectul d (isDirectory)
       files = d.list();                   // apel metoda ptr obiectul d (list)
       System.out.println ( files.length);
     }
}
```

Una din cele mai folosite clase în Java este clasa *String*, care poate genera obiecte ce contin fiecare un sir de caractere. Clasa *String* contine metode pentru cãutarea într-un sir, pentru extragere de subsiruri, pentru comparatie de siruri si pentru anumite operatii de modificare a unui sir.

Obiectele sunt rezultatul instantierii unei clase. In Java instantierea se face numai la executie, folosind direct operatorul *new* (sau apelând o metodã "fabricã" de obiecte). Adresa unui obiect se memoreazã într-o variabilã referintã de tipul clasei cãreia apartine obiectul. Exemple:

```
String  fname = new String ("test.java ");       // creare sir cu un nume de fisier
int p = fname.indexOf ('.');                     // pozitia primului punct din nume
String fext = fname.substring (p+1);             // creare sir cu extensia numelui
```

In exemplul de mai sus, metoda "indexOf" se aplicã obiectului cu adresa în variabila "fname" si are ca rezultat un întreg, iar metoda "substring", aplicatã aceluiasi obiect are ca rezultat un alt obiect, memorat la adresa din variabila "fext".

Variabilele de tipuri clasã reprezintã singura posibilitate de acces la obiectele Java si ele corespund variabilelor referintã din C++.

In Java gestiunea memoriei dinamice este automatã iar programatorul nu trebuie sã aibã grija eliberãrii memoriei alocate pentru obiecte. Teoretic, memoria ocupatã de un obiect este eliberatã atunci când obiectul respectiv devine inaccesibil si inutilizabil, dar momentul exact al recuperãrii memoriei nu poate fi precizat si depinde de modul de gestiune a memoriei.

Este posibilã apelarea directã a colectorului de resturi de memorie (“garbage collector”), dar nu se practicã decât foarte rar.

Pentru utilizarea mai eficientã a memoriei si pentru reducerea timpului de executie se recomandã sã nu se creeze obiecte inutile, atunci când existã alte posibilitãti. De exemplu, o variabilã de un tip clasã care va primi ulterior rezultatul unei functii nu va fi initializatã la declarare cu altceva decât cu constanta *null*. Exemplu:

```
RandomAccessFile  f = new RandomAccessFile ("date.txt","r");
String line=null;            //  nu:   line = new String();
...
line = f.readLine();         // citeste o linie din fisierul f
```

O altã situatie este cea în care un vector de obiecte trebuie initializat cu un acelasi obiect (de fapt cu o referintã la un obiect unic):

```
Integer zero = new Integer (0);              // un obiect de tip Integer
Integer count[ ]= new Integer [n];           // un vector cu elem. de tip Integer
for (int i=0; i<n;i++)  count[i] = zero;     // nu:  count[i]= new Integer(0);
```

O a treia situatie frecventã este la construirea unui obiect pe baza altor obiecte; de obicei este suficientã retinerea adresei obiectului primit ca parametru de constructor si nu trebuie creat un nou obiect. Exemplu:

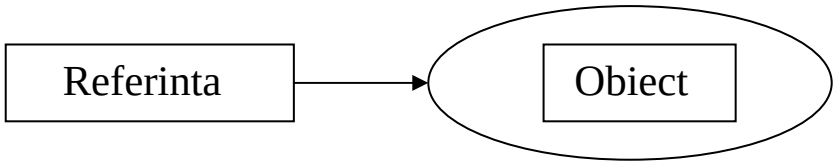
```
class Elev {
  String nume;  Float medie;                          // datele clasei
  public Elev ( String unnume, float med) {       // constructor al clasei Elev
      nume = unnume;                              //  nu:  nume = new String(unnume);
      medie =  new Float (med);
   }
    . . .    // metode ale clasei Elev
}
```

Variabile referintã la un tip clasã

Variabilele Java pot fi: variabile de un tip primitiv (*char, int, float, boolean* etc) sau variabile de un tip clasã (sau de un tip vector), care sunt variabile referintã. Nu se pot defini referinte la tipuri primitive sau parametri referintã de un tip primitiv.

Variabila care primeste rezultatul operatorului *new* nu contine chiar obiectul ci este o referintã la obiectul creat. O referintã la un tip T este de fapt un pointer la tipul

T care se foloseste ca si cum ar fi o variabilã de tipul T. Indirectarea prin variabila referintã este realizatã automat de compilator, fãrã a se folosit un operator .



Simpla declarare a unei variabile de un tip clasã nu antreneazã automat crearea unui obiect. Compilatorul Java verificã si anuntã utilizarea unei variabile care nu a fost initializatã. Secventa urmãtoare va provoca o eroare la compilare :

```
String nume;
System.out.println (nume);        // utilizare variabilã neinitializatã
```

Variabilele declarate în functii nu sunt initializate de compilator, dar variabilele declarate în afara functiilor sunt initializate automat cu zero sau cu *null*. Exemplu de eroare care nu este semnalatã la compilare si produce o exceptie la executie:

```
import java.util.*;
class Eroare {
  static Vector v;
  public static void main  (String arg[]) {
    System.out.println (v.size());              // NullPointerException
  }
}
```

Pentru elementele unui vector intrinsec compilatorul nu poate stabili dacã au fost sau nu initializate si se produce exceptie la executie. Exemplu:

```
String  tab[ ] = new String[10];          // tab[i]=null  in mod implicit
int n=tab[0].length();                    // NullPointerException
```

Utilizarea operatorului de comparatie la egalitate "==" între douã variabile referintã are ca efect compararea adreselor continute în cele douã variabile si nu compararea datelor adresate de aceste variabile. Exemplu de eroare:

```
String linie=f.readLine(); ...            // citeste o linie din fisierul f
if (linie == ".") break;                  // incorect, se compara adrese !
```

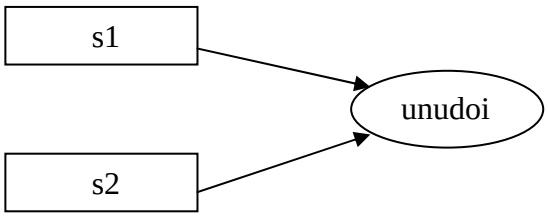
Comparatia la egalitate între obiecte Java se face fie prin metoda "equals" (de tip *boolean*), fie prin metoda "compareTo" (de tip *int*).Exemplu :

```
if (linie.equals (".")) break;     // sau  if (linie.compareTo(".")==0) break;
```

Metoda “equals” existã în orice clasã Java dar metoda “compareTo” existã numai în clasele cu obiecte “comparabile”, cum sunt clasele *String, Integer, Float, Date* etc.

Operatorul de atribuire se poate folosi numai între variabile referintã de acelasi tip sau pentru atribuirea constantei *null* la orice variabilã referintã. Efectul atribuirii între douã variabile referintã este copierea unei adrese si nu copierea unui obiect. Atribuirea între variabile referintã duce la multiplicarea referintelor cãtre un acelasi obiect. Exemplu:

```
StringBuffer s2, s1=new StringBuffer (“unu”);
s2=s1;                          // adresa sirului “unu”
s2.append (”doi”);
System.out.println (s1);        // scrie:   unudoi
```



Copierea datelor dintr-un obiect într-un alt obiect de acelasi tip se poate face:

a) Prin construirea unui nou obiect pe baza unui obiect existent (dacã existã constructor):

```
String s1 = “unu”;   String s2 = new String (s1);
```

b) Prin construirea unui nou obiect care preia datele din vechiul obiect (daca existã metode de extragere a datelor dintr-un obiect). Exemplu:

```
Integer  m1 = new Integer(1);
Integer m2 = new Integer ( m1.intValue());
```

c) Folosind metoda generalã “clone” (mostenitã de la clasa *Object*), dar numai pentru obiecte din clase care implementeazã interfata *Clonable*. Exemplu:

```
Vector a, b;                    // referinte la doi vectori
b= (Vector) a.clone();          // creare b si copiere date din a în b
```

In metoda “clone” se alocã memorie pentru un nou obiect si se copiazã datele din obiectul vechi în noul obiect (rezultat al metodei “clone”). Copierea este superficialã în sensul cã se copiazã variabile referintã (pointeri) si nu datele la care se referã acele variabile. Un rezultat nedorit este aparitia unor obiecte cu date comune.

Clasele JDK care contin variabile de un tip referintã au metoda “clone” redefinitã pentru a face o copiere “profundã” a datelor din obiectul clonat în obiectul clonã.

Incercarea a utiliza o variabilã referintã cu valoarea *null* pentru apelarea unei metode cauzeazã exceptia *NullPointerException*. Exemplu:

```
String s=null;   int len = s.length();        // exceptie  !
```

Elementele unui vector de obiecte sunt initializate automat cu *null* la alocarea de memorie, iar prelucrarea unui vector completat partial poate produce exceptia de utilizare a unui pointer (referintã) cu valoarea *null*. Exemplu:

```
Object a[] = new Object [10] ;      // 10 valori null
a[0]="unu"; a[1]="doi"; a[2]="trei";
Arrays.sort(a);                     // exceptie aruncata de functia “sort”
```

Metoda “clone” de copiere a unui vector intrinsec nu copiazã si valorile null, retinând în vectorul clonã numai elementele nenule. Exemplu:

```
Object [ ] b = new Object[10];
 b = (Object[]) a.clone();
Arrays.sort (b);       // nu produce exceptie
```

Argumente de functii de tip referintã

In Java, ca si în C, transmiterea unui parametru efectiv la apelarea unei functii se face prin valoare, adicã se copiazã valoarea parametrului efectiv în parametrul formal corespunzãtor, înainte de executia instructiunilor din functie. Argumentele de un tip primitiv nu pot fi modificate de o metodã Java, deci o metodã nu poate transmite mai multe rezultate de un tip primitiv, dar acest lucru nici nu este necesar. Argumentele unei metode sunt de obicei date initiale, iar efectul metodei este modificarea variabilelor clasei si nu modificarea argumentelor.

In cazul parametrilor de un tip clasã (parametri referintã) se copiazã adresa obiectului din functia apelantã în parametrul formal si nu se copiazã efectiv obiectul.

Prin copierea adresei unui obiect în parametrul formal corespunzãtor apar referinte multiple la un acelasi obiect (multiplicarea referintelor). O functie nu poate trasmite în afarã adresa unui obiect creat în functie printr-un parametru referintã. Exemplu de functie fãrã efect în afara ei:

```
      // metoda statica pentru trecere sir in litere mari - gresit !!!
 static void toUpper (String t) {
   t = t.toUpperCase();                // se creeaza un nou obiect, cu alta adresa
 }
```

Aici se creeazã un obiect *String* prin metoda “toUpperCase”, iar adresa sa este memoratã în variabila localã “t” (care continea initial adresa sirului dat). Un obiect creat într-o functie trebuie transmis ca rezultat al functiei. Exemplu:

```
static String toUpper (String s) {
  return  s.toUpperCase();
}
```

O functie poate modifica un obiect a cãrui adresã o primeste ca argument numai dacã în clasa respectivã existã metode pentru modificarea obiectelor.

Avantajele si riscurile obiectelor modificabile trasmise ca argumente pot fi ilustrate prin clasa *BitSet*, pentru multimi de întregi realizate ca vectori de biti. Diferenta a douã multimi A-B poate fi realizatã prin functia urmãtoare:

```
public static void minus (BitSet a, BitSet b) {
   for (int i=0;i<b.size();i++)
     if (b.get(i))                    // daca b[i] este 1
        a.clear(i);                        // atunci se face a[i]=0
 }
```

Dacã vrem sã pãstrãm multimea “a” atunci vom transmite o copie a ei

```
BitSet aux= (BitSet) a.clone();              // copiaza pe a in aux
minus (aux,b);                               // aux = aux-b
```

In exemplul urmãtor diferenta A-B se obtine pe baza unor operatii existente: A-B = A / (A*B) unde A/B este diferenta simetricã a multimilor A si B. Functia modificã în mod nedorit multimile primite ca argumente din cauza multiplicãrii referintelor.

```
public static BitSet minus (BitSet a, BitSet b) {
  BitSet c=a, d=a;              // c , d si a se refera la acelasi obiect
  c.and(b);                  //  c= a*b  dar s-a modificat si a !
  d.xor(c);                  //  d= d / c   dar s-a modificat si a !
  return d;                  //  d este multimea vidã !
}
```

Urmeazã o variantã corectã pentru calculul diferentei a douã multimi de tip *BitSet*:

```
public static BitSet minus (BitSet a, BitSet b) {
  BitSet c=new BitSet();
  BitSet d=new BitSet();
  c.or(a); c.and(b);            // c=a*b
  d.or(a); d.xor(c);            // d=b-a
  return d;
}
```

Clase cu obiecte nemodificabile

Clasele *String, Integer, Float* s.a. nu contin metode pentru modificarea datelor din aceste clase, deci o functie care primeste o referintã la un astfel de obiect nu poate modifica acel obiect. In acest fel se protejeazã obiectul transmis ca argument fatã de modificarea sa nedoritã de cãtre functia care îl foloseste. Obiectele din clase fãrã metode de modificare a datelor (clase “read-only”) se numesc obiecte nemodificabile (“immutable objects”). Clasa *String* este o clasã read-only si finalã, deci nu se poate extinde cu metode de modificare a vectorului de caractere continut în fiecare obiect.

Clasa *StringBuffer* a fost creatã ca o clasã paralelã cu clasa *String*, dar care contine în plus metode pentru modificarea obiectului (sirului). Exemple de metode care modificã sirul continut într-un obiect de tip *StringBuffer*: append, insert, delete, setCharAt, setLength. Un obiect de tipul *StringBuffer* transmis unei functii ca argument poate fi modificat de cãtre functie. Variantã pentru functia “toUpper”:

```
static void toUpper (StringBuffer s) {
  String str= new String (s);
  s.replace (0,str.length(),str.toUpperCase());
}
```

Concatenarea de siruri este o operatie frecventã în Java. Metoda “println” folositã pentru afisarea pe ecran poate avea un singur argument de tip *String*. Pentru a scrie mai multe siruri acestea se concatenează într-un singur sir cu operatorul ‘+’.Exemplu:

```
System.out.println ( “x= “ + x);          //  x de orice tip
```

Intr-o expresie cu operatorul binar ‘+’, dacã unul din operanzi este de tip *String*, atunci compilatorul Java face automat conversia celuilalt operand la tipul *String* (pentru orice tip primitiv si pentru orice tip clasã care redefineste metoda “toString”). Aceastã observatie poate fi folositã si pentru conversia unui numãr în sir de caractere, ca alternativã a utilizãrii metodei “valueOf” din clasa *String*. Exemplu:

```
float x = (float) Math.sqrt(2);
String str = ""+x;                        // sau    str = String.valueOf(x);
```

O instructiune de forma a=a+b; cu “a” si “b” de tip *String* este tratatã de compilator astfel: se transformã obiectele a si b în obiecte de tip *StringBufer*, se apeleazã metoda “append” si apoi creeazã un obiect *String* din obiectul *StringBuffer* rezultat din concatenare:

```
// secventa echivalentã cu a=a+b;
String a=“unu”, b=“doi”;
StringBuffer am= new StringBuffer (a),  bm= new StringBuffer (b);
am.append(bm);  a= new String (am);
```

Dacã trebuie sã facem multe concatenãri de siruri este preferabil sã se foloseascã direct metoda “append” din clasa *StringBuffer*. Exemplu:

```
public static String arrayToString ( int a[ ]) {
  StringBuffer aux = new StringBuffer(”[”);
  int n =a.length;
  for (int i=0;i<n-1;i++)
     aux.append (a[i] + ”,”);
  return new String (aux.append (a[n-1] +”]”) ) ;
}
```

Eliminarea unui subsir dintr-un sir se poate face folosind metoda “delete” din

clasa *StringBuffer* sau cu metode ale clasei *String*. Exemplu:

```
    // sterge caractere dintre pozitiile “from” si “to” din sirul s
   static String delete1 (String s, int from, int to ) {
      if ( from > to || from < 0 || to > s.length() )
         return s;                                  // s  nemodificat !
      return  s.substring(0,2) + s.substring(5);
    }
     // variantã cu StringBuffer
    static String delete2 (String s, int from, int to ) {
      StringBuffer sb = new StringBuffer(s);
      sb.delete (from,to);          // exceptie daca argumente incorecte !
      return sb.toString();
    }
```

De observat cã trecerea de la tipul *String* la tipul *StringBuffer* se poate face numai printr-un constructor, dar trecerea inversã se poate face prin metoda “toString”, iar aceste transformãri pot fi necesare pentru cã în clasa StringBuffer nu se regãsesc toate metodele din clasa *String*. De exemplu, metodele “indexOf” si “lastIndexOf” pentru determinarea pozitiei unui caracter sau unui subsir într-un sir (supradefinite în clasa *String*) nu existã în clasa *StringBuffer*.

Metoda “toString” existã în toate clasele ce contin date si produce un sir cu datele din obiectul pentru care se apeleazã (face conversia de la tipul datelor din obiect la tipul *String*).

Operatii cu siruri de caractere

Operatiile cu siruri sunt prezente în multe aplicatii Java si pot ilustra utilizarea de metode ale obiectelor si de metode ale claselor (statice).

O problemã uzualã în programare este extragerea de cuvinte (“tokens”), ce pot fi separate între ele prin unul sau mai multe caractere cu rol de separator, dintr-un text dat. Solutia uzualã creeazã un obiect analizor lexical (din clasa *StringTokenizer*) si apeleazã metode ale acestui obiect ("nextToken" = urmãtorul cuvânt):

```
import java.util.* ;
class Tokens {
  public static void main ( String[ ] args) {
     String text = new String ("unu doi,  trei.  patru;   cinci");      // sirul analizat
     String tokens[ ] = new String[100];           // vector de cuvinte
     StringTokenizer st = new StringTokenizer (text, " ,;.\t\n");   // separatori
     int k=0;                                       // numara cuvinte
      while (st.hasMoreTokens()) {                  // daca mai sunt cuvinte in sirul analizat
        String token = st.nextToken();              // extrage urmatorul cuvant din linie
        tokens[k++]=token;                          // memoreaza cuvant
     }
  }
}
```

Solutia Java 1.4 foloseste expresii regulate (metoda "split") pentru analiza textului:

```
class Tokens {
  public static void main ( String[ ] args) {
    String text = new String ("unu doi,  trei.  patru;   cinci");      // sirul analizat
    String tokens[] = text.split ("[,;.\t\n]+");       // argument expresie regulatã
  }
}
```

O expresie regulatã ("regular expression") este un sir de caractere cu rol de sablon ("pattern") pentru o multime de siruri care se "potrivesc" cu acel sablon. Expresiile regulate permit cãutarea de siruri, înlocuirea de siruri si extragerea de subsiruri dintr-un text (textul este obiect de tip "String").

Majoritatea operatiilor care folosesc expresii regulate se pot realiza în douã moduri:

a) Folosind metode noi din clasa "String".

b) Folosind metode ale claselor "Pattern" si "Matcher" din "java.util.regex".

Principalele metode din clasa "String" pentru lucrul cu expresii regulate sunt:

```
public boolean matches(String regex):
public String[] split(String regex)
public String replaceFirst(String regex, String replacement)
public String replaceAll(String regex, String replacement)
```

Metoda "matches" are rezultat "true" daca sirul pentru care se aplicã metoda se potriveste cu sablonul "regex". Metoda "split" creeazã un vector cu toate sirurile extrase din textul pentru care se aplicã metoda, siruri separate între ele prin siruri care se potrivesc cu sablonul "regex".

Metode mai importante din clasa *Pattern*:

```
static Pattern Pattern.compile(String regex);
boolean matches(String regex, String text);
String[] split(String text);
```

Metoda "compile" construieste un obiect de tip "Pattern" si retine sablonul primit într-un format intern (compilat). Metodele "matches" si "split" din clasa "Pattern" au acelasi efect ca si metodele cu acelasi nume din clasa "String". Exemplu:

```
String text = new String ("unu doi,  trei.  patru;   cinci");      // sirul analizat
Pattern p = Pattern.compile ("[,;.\t \n]+");
String tokens[] = p.split (text);          // argument expresie regulatã
```

Clasa *Matcher* contine metode pentru cãutare (find), potrivire (match), înlocuire (replaceAll, replaceFirst) s.a. un obiect de tipul *Matcher* se obtine apelând metoda “matcher” pentru un obiect *Pattern*. Exemplu de eliminare a comentariilor care încep cu caracterele “//” dintr-un text:

```
String regex="//.+\\n"   // o expresie regulatã
```

```
String text="linia1 // unu \nlinia2 // doi ";
Matcher m = Pattern.compile(regex).matcher(text);
String text2= m.replaceAll("\n");  // eliminare comentarii
```

In general, operatiile pe siruri se poate realiza mai compact cu expresii regulate, dar trebuie stãpânite regulile de formare ale acestor sabloane. Exemplu de eliminare a marcajelor HTML sau XML (‘tags”) dintr-un text:

```
String text=" <a>111<b> 222 </b> 333 </a> "        // sir cu marcaje
  // fara expresii regulate
textb = new StringBuffer(text);
while ((p1=text.indexOf('<',p2)) >=0 ) {        // p1=pozitia car. ‘<’
  if ( (p2= text.indexOf('>',p1+1)) > 0)          // p2 = pozitia car. ‘>’
     textb.delete(p1,p2+1);
}
  // cu expresii regulate
String regex="<[^<>]*>";              // orice sir incadrat de < si >
String text2= text.replaceAll(regex,"");
```

Clase si obiecte Java în faza de executie

Pentru fiecare clasã încãrcatã în masina virtualã Java este creat automat câte un obiect de tip *Class*, cu informatii despre clasa asociatã (metadate). Obiecte *Class* sunt create automat si pentru interfete, clase abstracte si vectori intrinseci Java.

Prin “reflectie” (“reflection”) se întelege obtinerea de informatii despre o clasã sau despre un obiect în faza de executie (este “reflectatã” starea masinii virtuale). In plus, se pot crea si modifica dinamic obiecte în faza de executie.

Reflectia este asiguratã în principal de clasa numitã *Class*, dar si de alte clase din pachetul “java.lang.reflect”: *Constructor, Method, Field*, s.a.

O variabilã de tip *Class* contine o referintã la un obiect descriptor de clasã; ea poate fi initializatã în mai multe feluri:

- Folosind cuvântul cheie *class* (literalul *class*) ca si cum ar fi un membru public si static al clasei sau tipului primitiv :

```
Class  cF = Float.class, cS = Stiva.class,       // clase predefinite sau proprii
       cf = float.class, cv =void.class,          // tipuri primitive
       cN= Number.class, cI=Iterator.class ;    // clase abstracte si interfete
```

- Folosind metoda staticã “forName” cu argument nume de clasã (ca sir de caractere):

```
Class cF = Class.forName(“java.util.Float”), cf = Class.forName (“float”);
```

- Folosind metoda “getClass” pentru o variabilã de orice tip clasã (metoda “getClass” este mostenitã de la clasa *Object*, deci existã în orice clasã):

```
Float f = new Float (3.14);      Class cF = f.getClass();
```

Clasa *Class* contine metode care au ca rezultat numele clasei, tipul clasei (clasã sau interfatã sau vector), tipul superclasei, clasa externã, interfete implementate, tipul obiectelor declarate în clasã, numele câmpurilor (variabilelor clasei), numele metodelor clasei, formele functiilor constructor s.a.

Metoda “getName” are ca rezultat un sir ce reprezintã numele clasei al cãrui tip este continut într-un obiect *Class.* Exemplul urmãtor aratã cum se poate afisa tipul real al unui obiect primit ca argument de tipul generic *Object*:

```
void printClassName (Object obj) {
    System.out.println ( obj.getClass().getName());
}
```

Crearea de obiecte de un tip aflat în cursul executiei dar necunoscut la compilare (cu conditia ca tipul respectiv sã fi fost definit printr-o clasã, iar fisierul “class” sã fie accesibil la executie) se poate face simplu dacã în clasã existã numai un constructor fãrã argumente . Exemplu:

```
public static Object getInstance (String clsname) throws Exception {
   Class cl = Class.forName(clsname);     // clsname = nume clasa (complet)
   return cl.newInstance();
}
```

Deoarece putem obtine toti constructorii unei clase, cunoscând tipul argumentelor, se pot “fabrica” obiecte si apelând constructori cu argumente. Exemplu:

```
public static Object newObject (Constructor constructor, Object [ ] args) {
Object obj = null;
try {
   obj = constructor.newInstance(args);
 } catch (Exception e) { System.out.println(e); }
 return obj;
 }
      // utilizare
  Float x;
  Class cls = Float.class;
  Class[] argsCls = new Class[ ] {float.class};      // tip argumente constructor
  Constructor constr = cls.getConstructor(argsCls);  // obtinere constructor
  Object[] args = new Object[ ] { new Float(3.5) }; // argument efectiv ptr instantiere
  x =(Float) newObject (constr,args);
```

Prin reflectie un asamblor de componente dintr-un mediu vizual poate sã determine proprietãtile si metodele proprii unor obiecte, sã modifice proprietãtile acestor obiecte si sã genereze apeluri de metode între obiecte.

In rezumat, reflectia permite operatii cu clase si cu obiecte necunoscute la scrierea programului, dar care pot fi determinate dinamic, în cursul executiei.

4. Definirea de noi clase

Definirea unei clase în Java

Definirea unei clase se face prin definirea variabilelor si functiilor clasei. In Java toate metodele trebuie definite (si nu doar declarate) în cadrul clasei. Ordinea în care sunt definiti membrii unei clase (date si functii) este indiferentã. Este posibil ca o metodã sã apeleze o altã metodã definitã ulterior, în aceeasi clasã.

Datele clasei se declarã în afara metodelor clasei si vor fi prezente în fiecare obiect al clasei. Ele sunt necesare mai multor metode si sunt de obicei putine.

Majoritatea claselor instantiabile grupeazã mai multe functii în jurul unor date comune. Ca exemplu vom schita o definitie posibilã pentru o clasã ale cãrei obiecte sunt numere complexe (nu existã o astfel de clasã predefinitã în bibliotecile JDK):

```
public class Complex {
  // datele clasei
    private double re,im;                          // parte realã si parte imaginarã
 // metode ale clasei
   // adunare de numere complexe
  public void add ( Complex cpx) {
     re += cpx.re;
     im += cpx.im;
  }
   // scadere de numere complexe
  public void sub ( Complex cpx) {
     re = re - cpx.re;
     im = im - cpx.im;
  }
}
```

Se observã cã metodele unei clase au putine argumente, iar aceste argumente nu se modificã, ceea ce est tipic pentru multe clase cu date: metodele clasei au ca efect modificarea datelor clasei, iar argumentele sunt de obicei date initiale necesare pentru operatiile cu variabilele clasei. Acesta este si un avantaj al programãrii cu clase fatã de programarea proceduralã: functii cu argumente putine si nemodificabile.

Clasele de uz general se declarã *public* pentru a fi accesibile din orice alt pachet.

Toate variabilele numerice ale unei clase sunt initializate implicit cu zerouri si toate variabile referintã sunt initializate cu *null*.

Variabilele de interes numai pentru anumite metode vor fi definite ca variabile locale în functii si nu ca variabile ale clasei. Exemplu de variabilã localã unei metode:

```
public Complex conj ( ) {
    Complex c = new Complex();      // c este o variabilã localã
    c.re= re; c.im= -im;
    return c;
 }
```

Pentru utilizarea comodã a obiectelor clasei "Complex" mai sunt necesare functii pentru initializarea datelor din aceste obiecte (numite "constructori") si pentru afisarea datelor continute în aceste obiecte.

In Java clasele cu date nu contin metode de afisare pe ecran a datelor din obiectele clasei, dar contin o metodã cu numele “toString” care produce un sir de caractere ce reprezintã datele clasei si care poate fi scris pe ecran (cu metoda “System.out.println”) sau introdus într-un alt flux de date sau folosit de alte metode. Functia urmãtoare (metodã a clasei "Complex") trebuie inclusã în definitia clasei:

```
  // conversie în sir, pentru afisare
 public String toString ( ) {
  return ( "(" + re+ "," + im+ ")");
 }
```

Existenta metodei "toString" ne permite sã afisãm direct continutul unui obiect din clasa "Complex" astfel:

```
Complex c = new Complex();
System.out.println (c);          // scrie (0,0)
```

In plus, se poate afisa simplu continutul unei colectii de obiecte "Complex", pentru cã metoda "toString" a colectiei apeleazã metoda "toString" a obiectelor din colectie.

Redefinirea metodei "equals" în clasa "Complex" permite cãutarea unui obiect dat într-o colectie si alte operatii care necesitã comparatia la egalitate. Exemplu:

```
public boolean equals (Object obj) {
    Complex cobj = (Complex) obj;
    return re==cobj.re && im==cobj.im;
}
```

Functii constructor

Orice clasã instantiabilã are cel putin un constructor public, definit implicit sau explicit si apelat de operatorul *new* la crearea de noi obiecte. Un constructor este o functie fãrã tip si care are obligatoriu numele clasei din care face parte. Constructorii nu sunt considerati metode, deoarece nu pot fi apelati explicit (prin nume).

Variabilele oricãrei clase sunt initializate automat la încãrcarea clasei (cu valori zero pentru variabile numerice si *null* pentru variabile de orice tip clasã). Aceste valori sunt preluate de fiecare obiect creat, dacã nu se fac alte initializãri prin constructori.

Pentru a permite initializarea variabilelor clasei în mod diferit pentru fiecare obiect creat existã în clasele ce contin date unul sau mai multi constructori. Exemplu:

```
public class Complex {
    private double re, im;                    // re=parte realã ,im= parte imaginarã
    public Complex ( double x, double y) {
```

```
        re=x; im=y;
    }
    . . .   // metode ale clasei
}
```

Exemple de creare a unor obiecte de tipul “Complex” :

```
        Complex c1 = new Complex (2,3) ,  c2= c1.conj();                // c2 =(2,-3)
```

Dacã nu se defineste nici un constructor atunci compilatorul generează automat un constructor implicit, fãrã argumente si fãrã efect asupra variabilelor clasei (dar care apeleazã constructorul superclasei din care este derivatã clasa respectivã).

Este uzual sã existe mai multi constructori într-o clasã, care diferã prin argumente, dar au acelasi nume (un caz de supradefinire a unor functii). Exemplu:

```
    public Complex ( Complex c) {
        re=c.re; im=c.im;
    }
```

Este posibilã si supradefinirea unor metode ale claselor. De exemplu, putem defini douã metode de adunare la un complex, cu acelasi nume dar cu argumente diferite:

```
    // adunare cu un alt complex
  public void add ( Complex c) {  re += c.re;  im += c.im;  }
    // adunare cu un numar real
  public void add ( double x) { re = re + x;  }
```

Variabilele unei clase pot fi initializate la declararea lor, ceea ce are ca efect initializarea la încãrcarea clasei, aceeasi pentru toate obiectele clasei. Acest fel de initializare se practicã pentru variabile membru de tip clasã si pentru clase cu un singur obiect. Exemplu de initializare a unei variabile la declarare:

```
class TextFrame extends JFrame {
   JTextField t = new JTextField (10);          // initializata la incarcare
   public TextFrame () {                        // constructor clasa
     getContentPane().add(t,"Center");          // adauga camp text la fereastra
   }
   . . .                                        // alte metode
}
```

Intr-o aplicatie se va crea un singur obiect de tip "TextFrame", iar obiectul de la adresa "t" va avea întotdeauna mãrimea 10. Instructiunea din constructor se va executa o singurã datã, dar trebuie sã fie inclusã într-o functie. Crearea obiectului câmp text *JTextField* nu se face printr-o instructiune ci printr-o declaratie cu initializare.

In Java nu sunt permise argumente formale cu valori implicite si functii cu numãr variabil de argumente (la apelare), dar un constructor cu mai putine argumente poate apela un constructor cu mai multe argumente, dintre care unele au valori implicite.

Un constructor nu poate fi apelat explicit, ca si o metodã, iar atunci când este necesar acest apel (din aceeasi clasã) se foloseste variabila *this*. Exemplu:

```
// constructor cu un parametru din clasa Complex
public Complex (double re) {  this (re,0);  }   // apel constructor cu douã argumente

// constructor fara parametri din clasa Complex
public Complex () { this (0); }         // apel constructor cu un argument
```

Variabila "this"

Metodele nestatice dintr-o clasã actioneazã asupra unor obiecte din acea clasã. Atunci când se apeleazã o metodã pentru un anumit obiect, metoda primeste ca argument implicit o referintã la obiectul respectiv. Altfel spus, instructiunile urmãtoare :

```
int n = a.length();                 // lungimea sirului a
String b = a.substring (k);         // extrage subsir din pozitia k din sirul a
```

corespund instructiunilor urmãtoare din limbajul C:

```
int n = length(a);                  // lungimea sirului a
String b = substring (a,k);         // extrage subsir din pozitia k din sirul a
```

Acest argument implicit poate fi folosit in interiorul unei metode nestatice prin intermediul variabilei predefinite *this*. Cuvântul cheie *this* este o variabilã referintã care desemneazã în Java adresa obiectului curent ("acest obiect").

Definitia metodei "length" ar putea fi rescrisã folosind variabila *this* astfel:

```
public int length (this) {
    return this.count;     // "count" este o variabilã a clasei String
}
```

In mod normal nu trebuie sã folosim variabila *this* pentru referire la datele sau la metodele unui obiect, dar existã situatii când folosirea lui *this* este necesarã.

Un prim caz de folosire curentã a variabilei *this* este la definirea unui constructor cu argumente formale având aceleasi nume cu variabile ale clasei. Exemplu:

```
public Complex ( float re, float im) {
    this.re=re;  this.im=im;              // ptr. a distinge arg. formale de var. clasei
}
```

Astfel de situatii pot fi evitate prin alegerea unor nume de argumente diferite de numele variabilelor clasei, dar clasele JDK folosesc aceleasi nume (si variabila *this*) pentru cã argumentele unui constructor contin valori initiale pentru variabilele clasei.

Un alt caz de folosire a variabilei *this* este în instructiunea *return*, pentru metodele care modificã datele unui obiect si au ca rezultat obiectul modificat. Exemplu:

```
public final class StringBuffer {
   private char value[ ];                    // un vector de caractere
   private int count;                    // caractere efectiv folosite din vector
        // metode
   public  StringBuffer deleteCharAt (int index) {     // sterge caracterul din poz index
     System.arraycopy (value, index+1, value, index, count-index-1);
     count--;
     return this;                              // rezultatul este obiectul modificat de metodã
   }
        // ... alte metode ale clasei StringBuffer
}
```

Un caz particular este metoda "toString" din clasa "String":

```
  public String toString () {
     return this;
  }
```

Uneori un obiect îsi trimite adresa sa metodei apelate (metodã staticã sau dintr-un alt obiect). Exemple:

```
  // un obiect observat isi transmite adresa sa unui observator prin metoda “update”
for (int i = obs.length-1; i>=0; i--)                // din clasa "Observable"
          ((Observer) obs[i]).update(this, arg);   // apel metoda "Observer.update"

   // un vector isi transmite adresa sa metodei statice de sortare
class SortedVec extends Vector {
  public void addElement (Object obj) {      // adaugare element si ordonare
    super.addElement (obj);
    Collections.sort(this);         // ordonare obiect ptr care s-a apelat metoda “add”
  }
}
```

Variabila *this* nu poate fi folositã în metode statice.

Atribute ale membrilor claselor

Membrii unei clase, variabile si metode, au în Java mai multe atribute: tipul variabilei sau metodei (tip primitiv sau tip clasã), un atribut de accesibilitate (*public, protected, private*) plus atributele *static, final, abstract* (numai pentru metode).

Cu exceptia tipului, fiecare din aceste atribute are o valoare implicitã, folositã atunci când nu se declarã explicit o altã valoare. Cuvintele cheie folosite pentru

modificarea atributelor implicite se numesc "modificatori". De exemplu orice membru este nestatic si nefinal atunci când nu se folosesc modificatorii *static* sau *final*.

O variabilã staticã a clasei declaratã *final* este de fapt o constantã, nemodificabilã prin operatii ulterioare primei initializãri. Exemple de constante definite în clasa *Byte*:

```
public class Byte {
    public static final byte MIN_VALUE = -128;
    public static final byte MAX_VALUE = 127;
    public static final Class TYPE = Class.getPrimitiveClass(“byte”);
    ...  // constructori si metode clasa Byte
}
```

Se vede din exemplul anterior cã o variabilã staticã finalã poate fi initializatã si cu rezultatul unei functii, pentru cã initializarea se face la încãrcarea clasei.

Metodele unei clase instantiabile sunt de obicei nestatice, dar pot exista si câteva metode statice în fiecare clasã cu date. Exemplu din clasa *String*:

```
public static String valueOf(Object obj) {              // sir echivalent unui obiect
    return (obj == null) ? "null" : obj.toString();
}
```

O metodã staticã poate fi apelatã de o metodã nestaticã. Exemplu:

```
public String toString() {              // metoda din clasa "Integer"
    return String.valueOf(value); // metoda statica din clasa String
}
```

O metodã staticã (inclusiv functia *main*) nu poate apela o metodã nestaticã, pentru cã folosirea unei metode nesatice este conditionatã de existenta unui obiect, dar metoda staticã se poate folosi si în absenta unor obiecte. O metodã staticã poate apela un constructor. Exemplu:

```
public static Integer valueOf (String s) throws NumberFormatException {
    return new Integer(parseInt(s));                 // static int parseInt(String,int);
}
```

Parte din definitia unei metode este si clauza *throws*, care specificã ce fel de exceptii se pot produce în metoda respectivã si care permite compilatorului sã verifice dacã exceptiile produse sunt sau nu tratate acolo unde se apeleazã metoda. Exemplu de metodã din clasa *Integer*, pentru conversia unui sir de caractere într-un întreg :

```
public static int parseInt (String s) throws NumberFormatException {
  if (s == null)
      throw new NumberFormatException("null");
  . . .
}
```

Anumite situatii speciale apãrute în executia unei metode sunt uneori raportate prin rezultatul functiei (boolean) si nu prin exceptii. De exemplu, anumite functii de citire a unui octet dintr-un fisier au rezultat negativ la sfârsit de fisier, iar functia de citire a unei linii (într-un obiect *String*) are rezultat *null* la sfârsit de fisier.

Uneori este posibilã anticiparea si prevenirea unor exceptii. Exemplul urmãtor aratã cum se poate evita aparitia unei exceptii de iesire din limitele unui vector intrinsec, din cauza utilizãrii gresite a unui program Java:

```
    // program cu doua argumente în linia de comanda
public static void main ( String arg[ ]) {
   if ( arg.length < 2) {
        System.out.println (“usage: java copy input output”);
         return;
    }
. . .     // prelucrare argumente primite
}
```

O metodã declaratã *final* nu mai poate fi redefinitã într-o subclasã si ar putea fi implementatã mai eficient de cãtre compilator.

O metodã abstractã este o metodã cu atributul *abstract*, declaratã dar nedefinitã în clasa unde apare pentru prima datã. O clasã care contine cel putin o metodã abstractã este o clasã abstractã si nu poate fi instantiatã, deci nu se pot crea obiecte pentru aceastã clasã (dar se pot defini variabile de un tip clasã abstractã).

Metode declarate dar nedefinite în Java sunt si metodele "native" (cu atributul *native*), care se implementeazã într-un alt limbaj decât Java (de obicei limbajul C). Definitia lor depinde de particularitatile sistemului de calcul pe care se implementeazã metodele. Exemplu:

```
    // metoda System.arraycopy ptr copiere partiala vectori
public static native void arraycopy (Object s, int si, Object d, int di, int length);
```

Incapsularea datelor în clase

De obicei datele unei clase nu sunt direct accesibile utilizatorilor clasei, deci nu sunt accesibile functiilor din alte clase. Pe de o parte utilizatorii nu sunt interesati de aceste date (de exemplu, cel care foloseste o stivã nu are nevoie sã “vadã” vectorul stivã si alte detalii), iar pe de altã parte este mai sigur ca aceste date sã fie modificate numai de metodele clasei si nu de orice utilizator.

O clasã expune utilizatorilor sãi o interfatã publicã, care constã din totalitatea metodelor publice (accesibile) ale clasei si care aratã ce se poate face cu obiecte ale clasei respective. Documentatia unei clase prezintã numai interfata sa publicã, adicã constructorii publici, metodele publice (tip, nume, argumente) si variabilele public accesibile. Este posibilã modificarea implementãrii unei clase, cu mentinerea

interfetei sale. De exemplu, putem folosi o stivã realizatã ca listã înlãntuitã în locul unei stive vector, fãrã ca programele ce folosesc stiva sã necesite vreo modificare.

Atributul de accesibilitate se referã la drepturile unei clase asupra membrilor unei alte clase, si este legat de ceea ce se numeste "încapsulare" sau "ascunderea datelor". Toate obiectele unei clase au aceleasi drepturi, stabilite la definitia clasei.

Metodele unei clase pot folosi variabilele clasei, indiferent de atributul de accesibilitate al acestor variabile (*private, protected, public*). Metodele unei clase se pot apela unele pe altele, indiferent de ordinea definirii lor si de atributul de accesibilitate.

Atributul *public* se foloseste pentru functiile si/sau variabilele clasei ce urmeazã a fi folosite din alte clase. In general, metodele si constructorii unei clase se declarã *public*. Variabilele public accesibile se mai numesc si proprietãti ale obiectelor.

Datele unei clase au în general atributul *private* sau *protected*, iar accesul la aceste date este permis numai prin intermediul metodelor, publice, ale clasei respective.

Exista si cazuri, mai rare, în care nu se respectã principiul încapsulãrii iar variabilele unei clase sunt public accesibile. Este vorba de clase cu mai multe variabile, folosite relativ frecvent în alte clase si pentru care accesul prin intermediul metodelor ar fi incomod si ineficient. Exemplu din pachetul “java.awt”:

```
public class Rectangle {                // dreptunghi de incadrare figura geometrica
  public int x, y, width, height;       // coordonate:  colt, lãtime, înãltime
  public Rectangle (int x0, int y0, int w, int h) {
       x0=x; y0=y; width=w; height=h;
  }
  ... // alte metode
}
```

Variabilele cu atributul *protected* dintr-o clasã sunt accesibile pentru toate clasele derivate direct din A (indiferent de pachetul unde sunt definite) si pentru clasele din acelasi pachet cu ea.

Dacã nu se specificã explicit modul de acces la un membru al unei clase A atunci se considerã implicit cã el este accesibil pentru toate clasele definite în acelasi pachet cu A, numite uneori clase “prietene”. De exemplu, clasele *VectorEnumerator* si *Vector* sunt definite (în Java 1.0) în acelasi pachet si în acelasi fisier sursã, iar variabile cu atributul protected din clasa *Vector* sunt folosite direct în clasa *VectorEnumerator* (în Java 1.2 clasa enumerator este inclusã în clasa vector).

In continuare vom schita definitia unei clase pentru o listã înlãntuitã de numere întregi. Mai întâi trebuie definitã o clasã pentru un nod de listã, o clasã care poate sã nu aibã nici o metodã si eventual nici constructor explicit:

```
class Node {                                  // nod de lista inlantuita
   int val;                                  // date din nod
   Node leg;                                  // legatura la nodul urmator
   public Node (int v) { val=v; leg=null;}    // constructor
 }
```

Metodele clasei listã trebuie sã aibã acces direct la variabilele clasei “Node”, ceea ce se poate realiza în douã moduri:

- Clasele “Node” si “MyList” se aflã în acelasi pachet.
- Clasa “Node” este interioarã clasei “MyList”.

In ambele cazuri variabilele clasei “Node” pot sã nu aibã nici un atribut explicit de accesibilitate

Clasa “MyList” contine o variabilã de tip “Node” (adresa primului element din listã), care poate fi “private” sau “protected”, dacã anticipãm eventuale subclase derivate.

```
public class MyList {                          // lista inlantuita simpla (cu santinela)
  protected Node prim;                  // adresa primului element
  public MyList ( ) {
    prim = new Node(0);                 // creare nod sentinela (cu orice date)
  }
     // adaugare la sfirsit de lista
  public void add (int v) {             // adaugare intreg v la lista
     Node nou= new Node(v);             // creare nod nou cu valoarea v
     Node p=prim;                       // inaintare spre sfarsit de lista
     while (p.leg != null)
       p=p.leg;
     p.leg=nou;                         // nou urmeaza ultimului element
  }
      // sir ptr. continut lista
  public String toString () {
    String aux="";                  // rezultatul functiei toString
    Node p=prim.leg;                    // se pleaca de la primul nod din lista
    while ( p != null) {                // cat timp mai exista un element urmator
      aux=aux + p.val + "  ";           // intregul p.val se converteste automat in “String”
      p=p.leg;                          // avans la nodul urmator
    }
     return aux;                        // sirul cu toate valorile din lista
  }
}
```

O metodã a unei clase nu poate fi recursivã, cu argument modificabil de tipul clasei; solutia problemei este o functie staticã recursivã, apelatã de metodã. Exemplu de functii din clasa “MyList”:

```
     // cautare in lista
  public boolean contains (int x) {
     return find (prim,x);   // apel fct recursiva
  }
    // fct recursiva de cautare
  private boolean find (Node crt, int x) {
    if (crt==null)          // daca sfarsit de lista
       return false;        // atunci x negasit
    if ( x==crt.val)
        return true;        // x gasit
```

```
   return find (crt.leg,x);   // continua cautarea in sublista
 }
```

In exemplul anterior recursivitatea nu este justificatã, dar pentru alte clase (cu arbori, de exemplu) forma recursivã este preferabilã.

Structura unei clase Java

O clasã care poate genera obiecte contine în mod normal si date, deci variabile ale clasei, definite în afara metodelor clasei (de obicei înaintea functiilor).

Majoritatea claselor instantiabile au unul sau mai multi constructori publici folositi pentru initializarea obiectelor si apelati de operatorul *new*.

Metodele unei clase pot fi clasificate în câteva grupe:

- Metode care permit citirea datelor clasei (metode accesor).
- Metode care permit modificarea datelor clasei (si care lipsesc din anumite clase).
- Metode pentru operatii specifice clasei respective.
- Metode mostenite de la clasa *Object* si redefinite, pentru operatii generale, aplicabile oricãrui obiect Java (“equals”, “toString”, “hashCode”).

Conform unor cerinte mai noi ale standardului Java Beans, metodele de acces la variabile *private* au un nume care începe cu "get" si continuã cu numele variabilei, iar metodele de modificare a variabilelor au un nume care începe cu "set". Exemplu:

```
public class Complex {
  private double  re, im;          // parte reala si imaginara
        // metode de citire a datelor
    public float getReal ( )  { return re; }      // extrage parte reale
    public float getImag ( ) { return im; }      // extrage parte imaginara
        //  metode de modificare a datelor
    public void setReal (float x) { re =x; }         // modifica parte reala
    public void setImag (float y) { im=y; }         // modifica parte imaginara
        // complex conjugat
    public Complex conj () {
       return new Complex(re,-im);
    }
        // alte operatii cu obiecte de tip Complex
    . . .
        // metode din “Object” redefinite
    public boolean equals (Object obj) {. . . }
    public String toString () { . . . }
}
```

Clasa *String* contine un vector de caractere, dimensiunea sa si mai multe functii care realizeazã diverse operatii asupra acestui vector: cãutarea unui caracter dat în vector, extragerea unui subsir din întregul sir, compararea cu un alt sir etc. Exemplu:

```
public class String  {
        // variabilele clasei
   private char value[ ];                          // un vector de caractere
```

```
   private int count;                              // numar de caractere folosite
      // un constructor
    public String( char [ ] str) {
      count= str.length;  value= new char[count];
      System.arraycopy (str,0,value,0,count);
    }
        // metodele clasei
    public int length () {                         // lungime sir
      return count;
    }
    public String substring (int start, int end) { // extrage subsir dintre start si end
      int n=end-start+1;                           // nr de caractere intre start si end
      char b[ ] = new char[n];
      System.arraycopy (value,start,b,0,n);
      return new String (b);
   }
   public String substring (int pos) {             // extrage subsir din pozitia pos
       return substring (pos, length()-1);         // return this.substring (pos,length()-1);
   }
        // ... alte metode
}
```

A se observa existenta unor metode cu acelasi nume dar cu argumente diferite si modul în care o metodã (nestaticã) apeleazã o altã metodã nestaticã din clasã: nu se mai specificã obiectul pentru care se apeleazã metoda “substring” cu doi parametri, deoarece este acelasi cu obiectul pentru care s-a apelat metoda “substring” cu un singur parametru (obiectul *this* = chiar acest obiect).

O clasã mai poate contine constante si chiar alte clase incluse.

In limbajul C o declaratie de structurã este o definitie de tip care nu alocã memorie si de aceea nu este posibilã initializarea membrilor structurii. In Java o definitie de clasã creeazã anumite structuri de date si alocã memorie, ceea ce justificã afirmatia cã o clasã este un fel de sablon pentru crearea de obiecte de tipul respectiv. Definitia unei clase Java nu trebuie terminatã cu ‘;’ ca în C++.

Metode care pot genera exceptii

In antetul unor metode trebuie sã aparã clauza *throws*, dacã în aceste metode pot apare exceptii, a cãror tratare este verificatã de compilator. Intr-o metodã poate apare o exceptie fie datoritã unei instructiuni *throw*, fie pentru cã se apeleazã o metodã în care pot apare exceptii.

In cursul executiei unei metode sau unui constructor este posibil sã aparã o situatie de exceptie, iar metoda trebuie sã anunte mai departe aceastã exceptie. Semnalarea unei exceptii program înseamnã în Java crearea unui obiect de un anumit tip clasã (derivat din clasa *Exception*) si transmiterea lui în afara functiei, cãtre sistemul care asistã executia ("Runtime system"). Exemplu de metodã care poate produce exceptii:

```
// metoda din clasa String
```

```
public char charAt (int index) {
    if ((index < 0) || (index >= count))
       throw new StringIndexOutOfBoundsException(index);
    return value[index + offset];
   }
```

Instructiunea *throw* se foloseste pentru semnalarea unei exceptii într-o metodã si specificã un obiect “exceptie”(de obicei un obiect anonim, creat chiar în instructiune). Alegerea acestui cuvânt ("to throw"= a arunca) se explicã prin aceea cã la detectarea unei exceptii nu se apeleazã direct o anumitã functie (selectatã prin nume); functia care va trata exceptia (numitã "exception handler") este selectatã dupã tipul obiectului exceptie "aruncat" unui grup de functii.

Metoda "charAt" aruncã o exceptie care nu trebuie declaratã.

In exemplul urmãtor metoda "parseByte" aruncã o exceptie ce trebuie tratatã (de exemplu, prin repetarea citirii sirului primit), exceptie care trebuie anuntatã mai departe, pentru a se verifica tratarea ei.

```
        // metoda din clasa Byte
public static byte parseByte (String s, int radix) throws NumberFormatException {
    int i = Integer.parseInt(s, radix);
    if (i < MIN_VALUE || i > MAX_VALUE)
       throw new NumberFormatException();            // generează exceptie
    return (byte)i;
}
```

O metodã în care se poate produce o exceptie si care aruncã mai departe exceptia trebuie sã continã în definitia ei clauza *throws*. Cuvântul cheie *throws* apare în antetul unei metode, dupã lista de argumente si este urmat de numele unui tip exceptie (nume de clasã) sau de numele mai multor tipuri exceptie. In Java se considerã cã erorile semnalate de o metodã fac parte din antetul metodei pentru a permite compilatorului sã verifice dacã exceptia produsã este ulterior tratatã si sã oblige programatorii la tratarea anumitor exceptii. Exemplu de exceptie generatã de un constructor:

```
    public static FileReader fopen ( String filename) throws IOException {
        return new FileReader (filename);            // exceptie de fisier negãsit
    }
```

Orice functie (metodã) care apeleazã metoda “fopen” de mai sus trebuie fie sã arunce mai departe exceptia (prin clauza *throws*), fie sã o trateze (prin *try-catch*):

```
    public static fileCopy ( String src, String dst) throws IOException {
        FileReader fr = fopen (src);        // aici poate apare exceptie de I/E
        …
```

Pentru erori uzuale sunt predefinite o serie de tipuri exceptie, dar utilizatorii pot sã-si defineascã propriile tipuri exceptie, sub forma unor noi clase (derivate fie din clasa *Exception* fie din clasa *RuntimeException*).

Pentru erorile care nu permit continuarea programului vom prefera exceptiile derivate din clasa *RuntimeException* sau din clasa *Error*, care nu obligã programatorul la o anumitã actiune ("prindere" sau aruncare mai departe). Existã chiar pãrerea cã toate exceptiile ar trebui sã fie de acest fel, pentru simplificarea codului (prin eliminarea clauzei *throws* din antetul metodelor) si pentru a evita tratarea lor superficialã (prin interzicerea oricãrui mesaj la aparitia exceptiei).

Exceptiile aruncate de toate functiile (inclusiv de “main”) au ca efect terminarea programului dupã afisarea unui mesaj si a secventei de apeluri ce a dus la exceptie.

5. Derivare, mostenire, polimorfism

Clase derivate

Derivarea înseamnã definirea unei clase D ca o subclasã a unei clase A, de la care "mosteneste" toate variabilele si metodele publice. Subclasa D poate redefini metode mostenite de la clasa pãrinte (superclasa) A si poate adãuga noi metode (si variabile) clasei A. Tipul D este un subtip al tipului A. La definirea unei clase derivate se foloseste cuvântul cheie *extends* urmat de numele clasei de bazã. Exemplu:

```
    // vector ordonat cu inserare element nou
public class SortedVector extends Vector {
    // metoda redefinita ptr adaugare la vector ordonat
  public void addElement (Object obj) {      // adaugare element si ordonare
    int i=indexOf (obj);
    if (i < 0)
      i=-i-1;
    insertElementAt(obj,i);
  }
    // metoda redefinita: cautare binara in vector ordonat
  public int indexOf (Object obj) {          // cauta in vector un obiect
    return Collections.binarySearch(this,obj);
  }
}
```

In Java se spune cã o subclasã extinde functionalitatea superclasei, în sensul cã ea poate contine metode si date suplimentare. In general o subclasã este o specializare, o particularizare a superclasei si nu extinde domeniul de utilizare al superclasei.

Cuvântul *super*, ca nume de functie, poate fi folosit numai într-un constructor si trebuie sã fie prima instructiune (executabilã) din constructorul subclasei. Exemplu:

```
public class IOException extends Exception {
   public IOException() { super(); }
   public IOException(String s) { super(s); }
}
```

Principala modalitate de specializare a unei clase este redefinirea unor metode din superclasã. Prin redefinire ("override") se modificã operatiile dintr-o metodã, dar nu si modul de utilizare al metodei.

De multe ori, subclasele nu fac altceva decât sã redefineascã una sau câteva metode ale superclasei din care sunt derivate. Redefinirea unei metode trebuie sã pãstreze "amprenta" functiei, deci numele si tipul metodei, numãrul si tipul argumentelor .

Majoritatea claselor Java care contin date redefinesc metoda "toString" (mostenitã de la clasa *Object*), ceea ce permite conversia datelor din clasã într-un sir de caractere

si deci afisarea lor. Nu se poate extinde o clasã finalã (cu atributul *final*) si nu pot fi redefinite metodele din superclasã care au unul din modificatorii *final, static, private*.

O metodã redefinitã într-o subclasã "ascunde" metoda corespunzãtoare din superclasã, iar o variabilã dintr-o subclasã poate "ascunde" o variabilã cu acelasi nume din superclasã. Apelarea metodei originale din superclasã nu este în general necesarã pentru un obiect de tipul subclasei, dar se poate face folosind cuvântul cheie "super" în locul numelui superclasei. Exemplu:

```
    // vector ordonat cu adaugare la sfarsit si reordonare
public class SortedVector extends Vector {
  public void addElement (Object obj) {              // adaugare element si ordonare
    super.addElement (obj);                  // apel metoda “addElement” din clasa Vector
    Collections.sort(this);
  }
    // metoda mostenita automat dar interzisa in subclasa
  public void insertElementAt (Object obj, int pos) {   // ptr a nu modifica ordinea
    throw new UnsupportedOperationException();
  }
}
```

O subclasã se poate referi la variabilele superclasei fie direct prin numele lor, dacã variabilele din superclasã sunt de tip *public* sau *protected*, fie indirect, prin metode publice ale superclasei. Exemplu:

```
public class Stack extends Vector {
. . .
    public boolean empty() {       // test de stiva goala
        return size()==0;         // sau return elementCount==0;
    }
}
```

Derivarea ca metodã de reutilizare

Derivarea este motivatã uneori de necesitatea unor clase care folosesc în comun anumite functii dar au si operatii specifice, diferite în clasele înrudite. O solutie de preluare a unor functii de la o clasã la alta este derivarea.

O clasa derivatã "mosteneste" de la superclasa sa datele si metodele nestatice cu atributele *public* si *protected* . Clasa “SortedVector”, definitã anterior, mosteneste metodele clasei *Vector*: elementAt, indexOf, toString s.a. Exemplu:

```
    SortedVector a = new SortedVector();
    System.out.println ( a.toString());              // afisare continut vector
```

Chiar si metoda “addElement” redefinitã în clasa derivatã, refoloseste extinderea automatã a vectorului, prin apelul metodei cu acelasi nume din superclasã.

In Java atributele membrilor mosteniti nu pot fi modificate în clasa derivatã: o variabilã *public* din superclasã nu poate fi fãcutã *private* în subclasã.

Metodele si variabilele mostenite ca atare de la superclasã nu mai trebuie declarate în subclasã. In schimb, trebuie definite functiile constructor, metodele si datele suplimentare (dacã existã) si trebuie redefinite metodele care se modificã în subclasã.

O subclasã nu mosteneste constructorii superclasei, deci fiecare clasã are propriile sale functii constructor (definite explicit de programator sau generate de compilator). Nu se generează automat decât constructori fãrã argumente. Din acest motiv nu putem crea un obiect vector ordonat prin instructiunea urmãtoare:

```
SortedVector a = new SortedVector(n);   // nu exista constructor cu argument !
```

In Java constructorul implicit (fãrã argumente) generat pentru o subclasã apelează automat constructorul fãrã argumente al superclasei. Compilatorul Java generează automat o instructiune "super(); " înainte de prima instructiune din constructorul subclasei, dacã nu există un apel explicit si dacã nu se apelează un alt constructor al subclasei prin "this(...)".

Initializarea variabilelor mostenite este realizatã de constructorul superclasei, desi valorile folosite la initializare sunt primite ca argumente de constructorul subclasei. Este deci necesar ca un constructor din subclasã sã apeleze un constructor din superclasã. Apelul unui constructor din superclasã dintr-o subclasã se face folosind cuvântul cheie *super* ca nume de functie si nu este permisã apelarea directã, prin numele sãu, a acestui constructor. Exemplu:

```
public class SortedVector extends Vector {
  public SortedVector () {          // vector cu dimensiune implicita
     super();                       // initializari in superclasa
  }
  public SortedVector (int max) {// vector cu dimensiune specificata la instantiere
     super(max);                    // max folosit la alocarea de memorie in superclasa
  }
  . . .
```

Dacã se defineste un constructor cu argumente atunci nu se mai generează automat un constructor fãrã argumente de cãtre compilator (în orice clasã).

In Java apelul unei metode din constructorul superclasei se traduce prin apelarea metodei din subclasã si nu a metodei cu acelasi nume din superclasã, ceea ce uneori este de dorit dar alteori poate da nastere unor exceptii sau unor cicluri infinite.

In exemplul urmãtor clasa “SortVec” mentine o versiune ordonatã si o versiune cu ordinea cronologicã de adãugare a elementelor. La executia unei instructiuni SortVec vs = new SortVec(v); apare exceptia de utilizare a unui pointer nul în metoda “addElement” deoarece constructorul clasei *Vector* apelează metoda “addElement” din clasa “SortVec” iar variabila “v” nu este initializatã.

```
class SortVec extends Vector {
  private Vector v;        // vector cu ordinea de adaugare
  public SortVec () {
    super();    v= new Vector();
  }
```

```
  public SortVec ( Vector a) {
     super(a);                // apel constructor superclasa (produce exceptie !)
  }
  public void addElement (Object obj) {      // adaugare element si ordonare
    super.addElement (obj);          // adauga la vector ordonat dupa valori
    v.addElement(obj);               // adauga la vector ordonat cronologic
    Collections.sort(this);
  }
  . . .    // alte metode (si pentru acces la variabila v)
}
```

O variantã corectã a constructorului cu argument *Vector* aratã astfel:

```
  public SortVec ( Vector a) {
     v= new Vector(a);
     for (int i=0;i<v.size();i++)
        addElement(a.elementAt(i));
  }
```

In exemplul urmãtor clasa “MyQueue” (pentru liste de tip coadã) este derivatã din clasa “MyList” pentru liste simplu înlãntuite. Cele douã clase folosesc în comun variabila “prim” (adresa de început a listei) si metoda “toString”.

Metoda “add” este redefinitã din motive de performantã, dar are acelasi efect: adãugarea unui nou element la sfârsitul listei.

```
public class MyQueue extends MyList {          // Lista coada
  private Node ultim;                  // variabila prezenta numai in subclasa
  public MyQueue () {
    super();                          // initializare variabila “prim”
    ultim=prim;                       // adresa ultimului element din lista
  }
    // adaugare la sfirsit coada (mai rapid ca in MyList)
  public void add (Object v) {
     Node nou= new Node();
     nou.val=v;
     ultim.leg=nou;
     ultim=nou;
  }
      // eliminare obiect de la inceputul cozii
  public Object del () {
    if ( empty())
      return null;
    Nod p=prim.leg;           // primul nod cu date
    prim.leg=p.leg;
    if (empty())                  // daca coada este goala
      ultim=prim;
    return p.val;
  }
     // test daca coada goala
```

```
  public boolean empty() {
    return prim.leg==null;
  }
}
```

Din clasa “MyList” se poate deriva si o clasã stivã realizatã ca listã înlãntuitã, cu redefinirea metodei de adãugare “add”, pentru adãugare la început de listã, si cu o metodã de extragere (si stergere) obiect de la începutul listei. Ideea este cã atât coada cât si stiva sunt cazuri particulare de liste. Avantajele reutilizãrii prin extinderea unei clase sunt cu atât mai importante cu cât numãrul metodelor mostenite si folosite ca atare este mai mare.

Derivare pentru creare de tipuri compatibile

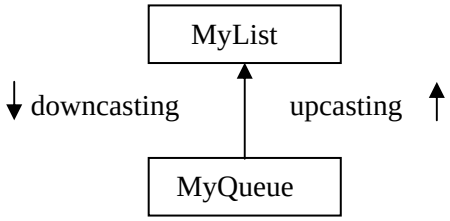
Definirea unei noi clase este echivalentã cu definirea unui nou tip de date, care poate fi folosit în declararea unor variabile, argumente sau functii de tipul respectiv. Prin derivare se creeazã subtipuri de date compatibile cu tipul din care provin.

Tipurile superclasã si subclasã sunt compatibile, în sensul cã o variabilã (sau un parametru) de un tip A poate fi înlocuit fãrã conversie explicitã cu o variabilã (cu un parametru) de un subtip D, iar trecerea de la o subclasã D la o superclasã A se poate face prin conversie explicitã (“cast”). Conversia de tip între clase incompatibile produce exceptia *ClassCastException.*

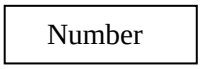
Exemple de conversii între tipuri compatibile:

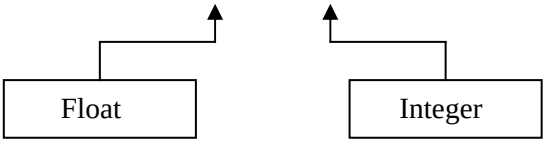
```
MyList a ; MyQueue q= new MyQueue();
a = q;                      //  upcasting
q = (MyQueue) a ;           //  downcasting
```

Tipuri compatibile sunt tipurile claselor aflate într-o relatie de descendentã (directã sau indirectã) pentru cã aceste clase au în comun metodele clasei de bazã. Conversia explicitã de la tipul de bazã la un subtip este necesarã pentru a putea folosi metodele noi ale subclasei pentru variabila de tip subclasã.



De remarcat cã douã subtipuri ale unui tip comun A nu sunt compatibile (de ex. tipurile *Integer* si *Float* nu sunt compatibile, desi sunt derivate din tipul *Number*). Trecerea între cele douã tipuri nu se poate face prin operatorul de conversie, dar se poate face prin construirea altui obiect.





Procesul de extindere sau de specializare a unei clase poate continua pe mai multe niveluri, deci fiecare subclasã poate deveni superclasã pentru alte clase, si mai specializate.

Posibilitatea de creare a unor tipuri compatibile si de utilizare a unor functii polimorfice constituie unul din motivele importante pentru care se foloseste derivarea. Aplicarea repetatã a derivării produce ierarhii de tipuri si familii de clase înrudite.

Putem constata în practica programării cu obiecte cã unele superclase transmit subclaselor lor putinã functionalitate (putine metode mostenite întocmai), deci motivul extinderii clasei nu este reutilizarea unor metode ci relatia specialã care apare între tipurile subclasei si superclasei.

In general se definesc familii de clase, unele derivate din altele, care sunt toate echivalente ca tip cu clasa de la baza ierarhiei. Un exemplu este familia claselor exceptie, derivate direct din clasa *Exception* sau din subclasa *RuntimeException*. Instructiunea *throw* trebuie sã continã un obiect de un tip compatibil cu tipul *Exception*; de obicei un subtip (*IOException, NullPointerException* s.a.).

O subclasã a clasei *Exception* nu adaugã metode noi si contine doar constructori.

Uneori superclasa este o clasã abstractã (neinstantiabilã), care nu transmite metode subclaselor sale, dar creeazã clase compatibile cu clasa abstractã.

Tipul superclasei este mai general decât tipurile subclaselor si de aceea se recomandã ca acest tip sã fie folosit la declararea unor variabile, unor argumente de functii sau componente ale unor colectii. In felul acesta programele au un caracter mai general si sunt mai usor de modificat.

Un exemplu din Java 1.1 este clasa abstractã *Dictionary*, care contine metode utilizabile pentru orice clasã dictionar, indiferent de implementarea dictionarului:
get, put, remove, keys, s.a. Clasa de bibliotecã *Hashtable* extinde clasa *Dictionary*, fiind un caz particular de dictionar, realizat printr-un tabel de dispersie. Putem sã ne definim si alte clase care sã extindã clasa *Dictionary*, cum ar fi un dictionar vector.
Atunci când scriem un program sau o functie vom folosi variabile sau argumente de tipul *Dictionary* si nu de tipul *Hashtable* sau de alt subtip. Exemplu de creare a unui dictionar cu numãrul de aparitii al fiecãrui cuvânt dintr-un vector de cuvinte:

```
public static Dictionary wordfreq ( String v[]) {
   Dictionary dic = new Hashtable();
   int cnt;                                  // contor de aparitii
   for (int i=0; i< v.length; i++) {
     Object key = v[i];                      // cuvintele sunt chei in dictionar
     Integer nr = (Integer) dic.get(key);    // valoarea asociata cheii key in dictionar
     if ( nr != null)                        // daca exista cheia in dictionar
       cnt = nr.intValue()+1;                // atunci se mareste contorul de aparitii
```

```
    else                                  // daca cheia nu exista in dictionar
      cnt =1;                             // atunci este prima ei aparitie
    dic.put (key, new Integer(cnt));      // pune in dictionar cheia si valoarea asociata
  }
  return dic;
}
```

Rezultatul functiei “wordfreq” poate fi prelucrat cu metodele generale ale clasei *Dictionary*, fãrã sã ne intereseze ce implementare de dictionar s-a folosit în functie. Alegerea unei alte implementãri se face prin modificarea primei linii din functie.
Exemplu de folosire a functiei:

```
public static void main (String arg[]) {
  String lista[ ] = {"a","b","a","c","b","a"};
  Dictionary d= wordfreq(lista);
  System.out.println (d);    // scrie  {c=1, b=2, a=3}
}
```

Utilizarea unui tip mai general pentru crearea de functii si colectii cu caracter general este o tehnicã specificã programãrii cu obiecte. In Java (si în alte limbaje) aceastã tehnicã a condus la crearea unui tip generic *Object*, care este tipul din care derivã toate celelalte tipuri clasã si care este folosit pentru argumentele multor functii (din clase diferite) si pentru toate clasele colectie predefinite (*Vector, HashTable*).

Clasa Object ca bazã a ierarhiei de clase Java

In Java, clasa *Object* (java.lang.Object) este superclasa tuturor claselor JDK si a claselor definite de utilizatori. Orice clasã Java care nu are clauza *extends* în definitie este implicit o subclasã derivatã direct din clasa *Object*. De asemenea, clasele abstracte si interfetele Java sunt subtipuri implicite ale tipului *Object*.

Clasa *Object* transmite foarte putine operatii utile subclaselor sale; de aceea în alte ierarhii de clase (din alte limbaje) rãdãcina ierarhiei de clase este o clasã abstractã. Deoarece clasa *Object* nu contine nici o metodã abstractã, nu se impune cu necesitate redefinirea vreunei metode, dar în practicã se redefinesc câteva metode.

Metoda “toString” din clasa *Object* transformã în sir adresa obiectului si deci trebuie sã fie redefinitã în fiecare clasã cu date, pentru a produce un sir cu continutul obiectului. Metoda “toString” permite obtinerea unui sir echivalent cu orice obiect, deci trecerea de la orice tip la tipul *String* (dar nu prin operatorul de conversie):

```
Date d = new Date();
String s =  d.toString();    // dar nu si :  String s = (String) d;
```

Metoda "equals" din clasa *Object* considerã cã douã variabile de tip *Object* sau de orice tip derivat din *Object* sunt egale dacã si numai dacã se referã la un acelasi obiect:

```
public boolean equals (Object obj) {
```

```
    return (this == obj);
  }
```

Pe de altã parte, un utilizator al clasei *String* (sau al altor clase cu date) se asteaptã ca metoda "equals" sã aibã rezultat *true* dacã douã obiecte diferite (ca adresã) contin aceleasi date. De aceea, metoda "equals" este rescrisã în clasele unde se poate defini o relatie de egalitate între obiecte. Exemplu din clasa “Complex”:

```
  public boolean equals (Object obj) {
    Complex cpx =(Complex) obj;
    if ( obj != null && obj instanceof Complex)
      if (this.re == cpx.re && this.im == cpx.im)
        return true;
    return false;
  }
```

Pentru a evita erori de programare de tipul folosirii metodei "String.equals" cu argument de tip "Complex", sau a metodei "Complex.equals" cu argument de tip "String" se foloseste în Java operatorul *instanceof*, care are ca operanzi o variabilã de un tip clasã si tipul unei (sub)clase si un rezultat boolean.

Redefinirea unei metode într-o subclasã trebuie sã pãstreze aceeasi semnãturã cu metoda din superclasã, deci nu se pot schimba tipul functiei sau tipul argumentelor. Din acest motiv, argumentul metodei "Complex.equals" este de tip *Object* si nu este de tip "Complex", cum ar pãrea mai natural. Dacã am fi definit metoda urmãtoare:

```
  public boolean equals (Complex cpx) { ... }
```

atunci ea ar fi fost consideratã ca o nouã metodã, diferitã de metoda "equals" mostenitã de la clasa *Object*. In acest caz am fi avut o supradefinire ("overloading") si nu o redefinire de functii.

La apelarea metodei "Complex.equals" poate fi folosit un argument efectiv de tipul "Complex", deoarece un argument formal de tip superclasã poate fi înlocuit (fãrã conversie explicitã de tip) printr-un argument efectiv de un tip subclasã. Existã deci mai multe metode "equals" în clase diferite, toate cu argument de tip *Object* si care pot fi apelate cu argument de orice tip clasã.

Metoda "equals" nu trebuie redefinitã în clasele fãrã date sau în clasele cu date pentru care nu are sens comparatia la egalitate, cum ar fi clasele flux de intrare-iesire.

O variabilã sau un argument de tip *Object* (o referintã la tipul *Object*) poate fi înlocuitã cu o variabilã de orice alt tip clasã, deoarece orice tip clasã este în Java derivat din si deci echivalent cu tipul *Object*. Exemplul urmãtor aratã cum un vector de obiecte *Object* poate fi folosit pentru a memora obiecte de tip *String* .

```
      // afisarea unui vector de obiecte oarecare
public static void printArray ( Object array [ ]) {
   for (int k=0; k<array.length; k++)
      System.out.println (array[k]);
 }
```

```
public static void main (String[ ] arg) {
  Object a[ ] = new Object [3] ;
  String s[ ] ={"unu", "doi", "trei"};
  for (int i=0; i<3; i++)
    a[i]= s[i];
  printArray (a);
}
```

Polimorfism si legare dinamicã

O functie polimorficã este o functie care are acelasi prototip, dar implementãri (definitii) diferite în clase diferite dintr-o ierarhie de clase.

Asocierea unui apel de metodã cu functia ce trebuie apelatã se numeste "legare" ("Binding"). Legarea se poate face la compilare (“legare timpurie”) sau la executie ("legare târzie" sau "legare dinamicã"). Pentru metodele finale si statice legarea se face la compilare, adicã un apel de metodã este tradus într-o instructiune de salt care contine adresa functiei apelate.

Legarea dinamicã are loc în Java pentru orice metodã care nu are atributul *final* sau *static*, metodã numitã polimorficã. In Java majoritatea metodelor sunt polimorfice. Metodele polimorfice Java corespund functiilor virtuale din C++.

Metodele “equals”, “toString” sunt exemple tipice de functii polimorfice, al cãror efect depinde de tipul obiectului pentru care sunt apelate. Exemple:

```
Object d = new Date();  Object f = new Float (3.14);
String ds = d.toString();    // apel  Date.toString()
String fs = f.toString(); // apel  Float.toString()
```

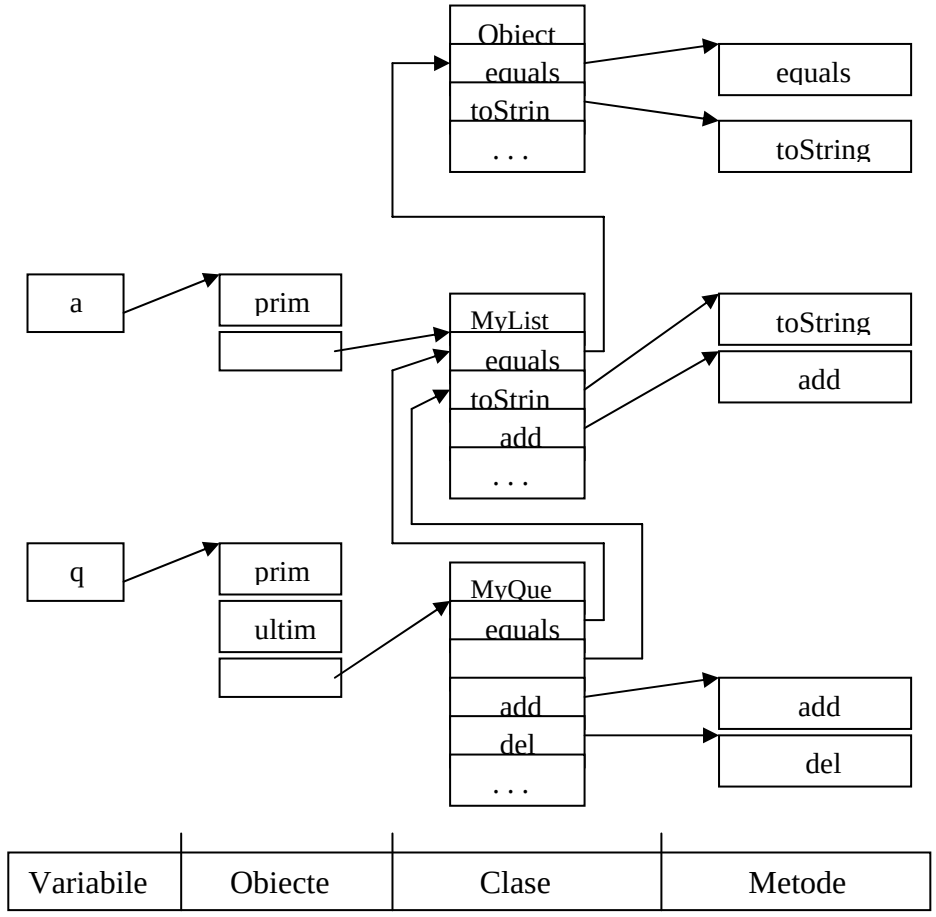
Mecanismul de realizare a legãrii dinamice este relativ simplu, dar apelul functiilor polimorfice este mai putin eficient ca apelul functiilor cu o singura formã. Fiecare clasã are asociat un tabel de pointeri la metodele (virtuale) ale clasei. Clasa MyQueue va avea o altã adresã (si o altã definitie) pentru functia "add" decât clasa MyList.

Fiecare obiect contine, pe lângã datele nestatice din clasã, un pointer la tabela de metode virtuale (polimorfice). Compilatorul traduce un apel de metodã polimorficã printr-o instructiune de salt la o adresã calculatã în functie de continutul variabilei referintã si de definitia clasei. Apelul “q.add” pleacã de la obiectul referit de variabile "q" la tabela metodelor clasei "MyQueue" si de acolo ajunge la corpul metodei "add" din clasa "MyQueue". Apelul “q.toString” conduce la metoda “toString” din superclasã (“MyList”) deoarece este o metodã mostenitã si care nu a fost redefinitã. Apelul “q.equals” merge în sus pe ierahia de clase si ajunge la definitia din clasa *Object*, deoarece nici o subclasã a clasei *Object* nu redefineste metoda “equals”.

Legãtura dinamicã se face de la tipul variabilei la adresa metodei apelate, în functie de tipul variabilei pentru care se apeleazã metoda. Fie urmãtoarele variabile:

```
MyList a, q;  a = new MyList();   q = new MyQueue();
```

Figura urmãtoare aratã cum se rezolvã, la executie, apeluri de forma a.add(), q.add(), q.toString(), q.equals() s.a.



Solutia functiilor polimorfice este specificã programãrii orientate pe obiecte si se bazeazã pe existenta unei ierarhii de clase, care contin toate metode (nestatice) cu aceeasi semnaturã, dar cu efecte diferite în clase diferite. Numai metodele obiectelor pot fi polimorfice, în sensul cã selectarea metodei apelate se face în functie de tipul variabilei pentru care se apeleazã metoda. O metodã staticã, chiar dacã este redefinitã în subclase, nu poate fi selectatã astfel, datoritã modului de apelare.

Alegerea automatã a variantei potrivite a unei metode polimorfice se poate face succesiv, pe mai multe niveluri. De exemplu, apelul metodei “toString” pentru un obiect colectie alege functia ce corespunde tipului de colectie (vector, lista înlãntuitã, etc); la rândul ei metoda “toString” dintr-o clasã vector apeleazã metoda “toString” a obiectelor din colectie, în functie de tipul acestor obiecte (*String*, *Integer*, *Date*, etc.).

O metodã finalã (atributul *final)* dintr-o clasã A, care este membrã a unei familii de clase, este o metodã care efectueazã aceleasi operatii în toate subclasele clasei A; ea nu mai poate fi redefinitã. Exemplu:

```
public final boolean contains (Object elem) {
  return indexOf (elem) >=0 ;
}
```

O metodã staticã poate apela o metodã polimorficã; de exemplu "Arrays.sort" apeleazã functia polimorficã "compareTo" (sau "compare"), al cãrei efect depinde de tipul obiectelor comparate, pentru a ordona un vector cu orice fel de obiecte.
Exemplu de functie pentru cãutare secventialã într-un vector de obiecte:

```
public static int indexOf (Object[] array, Object obj) {
    for (int i=0;i<array.length;i++)
        if ( obj.equals (array[i]))          // "equals" este polimorfica
            return i;                        // indice unde a fost gasit
    return -1;                               // negasit
}
```

Pentru functia anterioarã este esential cã metoda "equals" este polimorficã si cã se va selecta metoda de comparare adecvatã tipului variabilei "obj" (acelasi cu tipul elementelor din vectorul "array").

Structuri de date generice în POO

O colectie genericã este o colectie de date (sau de referinte la date) de orice tip, iar un algoritm generic este un algoritm aplicabil unei colectii generice.

O colectie genericã poate fi realizatã ca o colectie de obiecte de tip *Object* iar o functie genericã este o functie cu argumente formale de tip *Object* si/sau de un tip colectie genericã. Pentru a memora într-o colectie de obiecte si date de un tip primitiv (*int, double, boolean* s.a.) se pot folosi clasele "paralele" cu tipurile primitive: *Integer, Short, Byte, Float, Double,Boolean, Char*.

Primele colectii generice din Java au fost clasele *Vector, Hashtable* si *Stack* dar din Java 2 sunt mai multe clase si sunt mai bine sistematizate.

Cea mai simplã colectie genericã este un vector (intrinsec) cu componente de tip *Object:*

```
Object [ ] a = new Object[10];
for (int i=0;i<10;i++)
   a[i]= new Integer(i);
```

Clasa *Vector* oferã extinderea automatã a vectorului si metode pentru diferite operatii uzuale: cãutare în vector, afisare în vector, insertie într-o pozitie datã s.a.

Ceea ce numim o colectie de obiecte este de fapt o colectie de pointeri (referinte) la obiecte alocate dinamic, dar aceastã realitate este ascunsã de sintaxa Java. Exemplu de adãugare a unor siruri (referinte la siruri) la o colectie de tip *Vector*:

```
    Vector v = new Vector (10);        // aloca memorie pentru vector
    for (int k=1;k<11;k++)
       v.add ( k+"");          //  sau  v.add (String.valueOf(k));
```

Există posibilitatea de a trece de la un vector intrinsec la o clasă vector (metoda "Arrays.asList" cu rezultat *ArrayList*) si invers (metoda "toArray" din orice clasă colectie, cu rezultat Object[]).

Un alt avantaj al utilizării de clase colectie fată de vectori intrinseci este acela că se pot creea colectii de colectii, cu prelungirea actiunii unor functii polimorfice la subcolectii. De exemplu, metoda "toString" din orice colectie apelează repetat metoda "toString" pentru fiecare obiect din colectie (care, la rândul lui, poate fi o colectie).

La extragerea unui element dintr-o colectie generică trebuie efectuată explicit conversia la tipul obiectelor introduse în colectie, pentru a putea aplica metode specifice tipului respectiv. Exemplu de creare a unui vector de siruri, modificarea sirurilor si afisarea vectorului :

```
public static void main (String args[]) {
   Vector v = new Vector() ;
    for (int i=0;i<args.length;i++)
       v.add (args[i]);                              // adauga sir la vector
       // modificare vector
    for (int i=0;i<v.size();i++) {
       String str = (String)v.elementAt(i);          // sir din pozitia i
       v.setElementAt( str.toLowerCase(),i);         // trece sir in litere mici
    }
    System.out.println(v);                           // afisare vector
 }
```

In Java, ca si în C, sunt posibile expresii ce contin apeluri succesive de metode, fără a mai utiliza variabile intermediare. Exemplu:

```
    for (int i=0;i<v.size();i++)
       v.setElementAt ( (String)v.elementAt(i)).toLowerCase(), i);
```

Să considerăm o multime de obiecte realizată ca tabel de dispersie ("hashtable"). Tabelul de dispersie va fi realizat ca un vector de vectori, deci ca un obiect de tip *Vector*, având drept elemente obiecte de tip *Vector* (clasa *Vector* este si ea subclasă a clasei *Object*). O solutie mai bună este cea din clasa JDK *Hashtable*, în care se foloseste un vector de liste înlăntuite. Fiecare vector (sau listă) contine un număr de "sinonime", adică elemente cu valori diferite pentru care metoda de dispersie a produs un acelasi indice în vectorul principal. Metoda de dispersie are ca rezultat restul împărtirii codului de dispersie (produs de metoda "hashCode") prin dimensiunea vectorului principal. Numărul de sinonime din fiecare vector nu poate fi cunoscut si nici estimat, dar clasa *Vector* asigură extinderea automată a unui vector atunci când este necesar.

```
public class HSet extends Vector {  // tabel de dispersie ca vector de vectori
  private int n;
  public HSet (int n) {            // un constructor
    super(n);                      // alocare memorie ptr vector principal
    this.n=n;
    for (int i=0;i<n;i++)
      addElement (new Vector());  // aloca mem. ptr un vector de sinonime
  }
  public HSet () {                    // alt onstructor
    this(11);                      // dimensiune implicita vector principal
  }
     // apartenenta obiect la multime
  public boolean contains (Object el) {
    int k= (el.hashCode() & 0x7FFFFFFF) % n;     // poz. în vector principal (nr pozitiv)
    return ((Vector)elementAt(k)).contains(el);  // apel Vector.contains
  }
     // adaugare obiect la multime (daca nu exista deja)
  public boolean add (Object el) {
    int k = (el.hashCode() & 0x7FFFFFFF) % n;    // pozi. în vector principal (nr pozitiv)
    Vector sin = (Vector)elementAt(k);           // vector de sinonime
    if ( sin.contains (el))
      return false;
    sin.add (el);                                 // adauga la vector din pozitia k
    return true;
  }
     // numar de obiecte din colectie
  public int size() {
    int m=0;
    for (int k=0;k<n;k++)                         // ptr fiecare din cei n vectori
      m=m+((Vector)elementAt(k)).size();          // aduna dimensiune vector
    return m;
  }
     // conversie continut colectie în sir
  public String toString () {
    String s="";
    for (int k=0;k<n;k++) {
      Vector v=(Vector)elementAt(k);
      if (v !=null)
        s=s + v.toString()+"\n";
    }
    return s;
  }
```

Metodele polimorfice “toString”, “contains” si “size” din clasa “HSet” apeleazã metodele cu acelasi nume din clasa *Vector*. Redefinirea metodei “toString” a fost necesarã deoarece metoda “toString” mostenitã de la clasa *Vector* (care o mosteneste de la o altã clasã) foloseste un iterator, care parcurge secvential toate elementele vectorului, dar o parte din ele pot avea valoarea *null* si generează exceptii la folosirea lor în expresii de forma v.toString() .

Subliniem cã variabilele unei colectii generice au tipul *Object*, chiar dacã ele se referã la variabile de un alt tip (*Integer, String*, etc.); copilatorul Java nu “vede” decât tipul declarat (*Object*) si nu permite utilizarea de metode ale clasei din care fac parte obiectele reunite în colectie. Aceastã situatie este destul de frecventã în Java si face necesarã utilizarea operatorului de conversie *(tip)* ori de câte ori o variabilã de un tip mai general (tip clasã sau interfatã) se referã la un obiect de un subtip al tipului variabilei. Exemplu:

```
Object obj = v.elementAt(i);   // obiectul din vector poate avea orice tip
Float f = (Float) obj;          //  sau:     Float f = (Float) v.elementAt(i);
```

De remarcat cã astfel de conversii nu modificã tipul obiectelor ci doar tipul variabilelor prin care ne referim la obiecte (tipul unui obiect nu poate fi modificat în cursul executiei).

Incercarea de conversie între tipuri incompatibile poate fi semnalatã la compilare sau la executie, printr-o exceptie. Exemple:

```
Float f = new Float(2.5);  String s = (String) f;    // eroare la compilare
Object obj = f;   String s = (String) obj;    // exceptie la executie
```

In al doilea caz tipul *String* este subtip al tipului *Object* si deci compilatorul Java permite conversia (nu stie tipul real al obiectului referit de variabila “obj”).

6. Interfete si clase abstracte

C*lase abstracte în Java*

O superclasã este o generalizare a tipurilor definite prin subclasele sale; de exemplu, tipul *Number* este o generalizare a tipurilor clasã numerice (*Double, Float, Integer, Short, Byte*). Uneori superclasa este atât de generalã încât nu poate preciza nici variabile si nici implementãri de metode, dar poate specifica ce operatii (metode) ar fi necesare pentru toate subclasele sale. In astfel de cazuri superclasa Java este fie o clasã abstractã, fie o interfatã.

O metodã abstractã este declaratã cu atributul *abstract* si nu are implementare. Ea urmeazã a fi definitã într-o subclasã a clasei (interfetei) care o contine. Metodele abstracte pot apare numai în interfete si în clasele declarate abstracte.

O clasã care contine cel putin o metodã abstractã trebuie declaratã ca abstractã, dar nu este obligatoriu ca o clasã abstractã sã continã si metode abstracte. O clasã abstractã nu este instantiabilã, dar ar putea contine metode statice utilizabile.

In Java avem douã posibilitãti de a defini clase care contin doar metode statice si, ca urmare, nu pot fi instantiate (nu pot genera obiecte):

- Clase ne-abstracte, cu constructor *private*, care împiedicã instantierea;
- Clase abstracte, care nici nu au nevoie de un constructor

Spre deosebire de interfete, clasele abstracte pot contine si variabile.

O clasã abstractã este de multe ori o implementare partialã a unei interfete, care contine atât metode abstracte cât si metode definite. Exemplu:

```
public interface Collection {   // declara metode prezente in orice colectie
   int size();             // dimensiune colectie
   boolean isEmpty();      // verifica daca colectia este goala
   …                       // alte metode
}
public abstract class AbstractCollection implements Collection {
    public abstract int size();                          // metoda abstracta
    public boolean isEmpty ( return size()==0;}  // metoda implementata
    …
}
```

Scopul unei clase abstracte nu este acela de a genera obiecte utilizabile, ci de a transmite anumite metode comune pentru toate subclasele derivate din clasa abstractã (metode implementate sau neimplementate).

Derivarea dintr-o clasã abstractã se face la fel ca derivarea dintr-o clasã neabstractã folosind cuvântul *extend*. Exemplu:

```
public class MyCollection extends AbstractCollection {
    private Object [ ] array;                  // o colectie bazata pe un vector
    private int csize;
    public MyCollection (int maxsize) {
       array = new Object[maxsize];  csize=0;   // nr de obiecte in colectie
```

```
        }
        public int size() { return csize; }
        …       // alte metode
    }
```

O clasã abstractã poate face o implementare partialã, deci poate contine si metode neabstracte si/sau variabile în afara metodelor abstracte. Pentru a obtine clase instantiabile dintr-o clasã abstractã trebuie implementate toate metodele abstracte mostenite, respectând declaratiiile acestora (tip si argumente).

In bibliotecile de clase Java existã câteva clase adaptor, care implementeazã o interfatã prin metode cu definitie nulã, unele redefinite în subclase. Ele sunt clase abstracte pentru a nu fi instantiate direct, dar metodele lor nu sunt abstracte, pentru cã nu se stie care din ele sunt efectiv necesare în subclase. Exemplu de clasã adaptor:

```
public abstract class KeyAdapter implements KeyListener {
   public void keyTyped(KeyEvent e) { }        // la apasare+ridicare tasta
   public void keyPressed(KeyEvent e) { }      // numai la apasare tasta
   public void keyReleased(KeyEvent e) { }     // numai la ridicare tasta
}
```

O subclasã (care reactioneazã la evenimente generate de taste) poate redefini numai una din aceste metode, fãrã sã se preocupe de celalte metode nefolosite de subclasã:

```
    class KListener extends KeyAdapter {
        public void keyPressed(KeyEvent e) {
            char ch = e.getKeyChar();          // caracter generat de tasta apasata
            . . .                              // folosire sau afisare caracter ch
        }
    }
```

Interfete Java

O interfatã Java ar putea fi consideratã ca o clasã abstractã care nu contine decât metode abstracte si, eventual, constante simbolice. Totusi, existã deosebiri importante între interfete si clase abstracte în Java.

Metodele declarate într-o interfatã sunt implicit publice si abstracte.

De exemplu, interfata mai veche *Enumeration* contine douã metode comune oricãrei clase cu rol de enumerare a elementelor unei colectii :

```
public interface Enumeration {
    boolean hasMoreElements();          // daca mai sunt elemente in colectie
    Object nextElement();               // elementul curent din colectie
}
```

Enumerarea unor obiecte poate fi privitã ca o alternativã la crearea unui vector cu obiectele respective, în vederea prelucrãrii succesive a unui grup de obiecte.

O clasã (abstractã sau instantiabilã) poate implementa una sau mai multe interfete, prin definirea metodelor declarate în interfetele respective.

Dintre clasele care implementeazã aceastã interfatã, sunt clasele enumerator pe vector si enumerator pe un tabel de dispersie *Hashtable*. Ulterior s-au adãugat metode din aceastã interfatã si clasei *StringTokenizer*, care face o enumerare a cuvintelor dintr-un text:

```
public class StringTokenizer implements Enumeration {
 . . .
  public boolean hasMoreElements() {
     return hasMoreTokens();
  }
  public Object nextElement() {
     return nextToken();
  }
}
```

Utilizarea unei interfete comune mai multor clase permite unificarea metodelor si modului de utilizare a unor clase cu acelasi rol, dar si scrierea unor metode general aplicabile oricãrei clase care respectã interfata comunã. Exemplu de metodã de afisare a oricãrei colectii de obiecte pe baza unui obiect enumerator:

```
public static void print (Enumeration enum) {
   while (enum.hasMoreElements() )
       System.out.print (enum.nextElement()+ " , ");
   System.out.println();
}
```

Functia anterioarã poate fi aplicatã pentru orice colectie care are asociat un obiect enumerator (dintr-o clasã care implementeazã interfata *Enumeration*):

```
String text="a b c d e f ";
print ( new StringTokenizer(text));   // scrie  a , b , c , d , e , f ,
```

Nu pot fi create obiecte de un tip interfatã sau clasã abstractã, dar se pot declara variabile, argumente formale si functii de un tip interfatã sau clasã abstractã.

Prin definirea unei interfete sau clase abstracte se creeazã un tip de date comun mai multor clase deoarece o variabilã (sau un argument) de un tip interfatã poate fi înlocuitã fãrã conversie explicitã cu o variabilã de orice subtip, deci cu referinte la obiecte care implementeazã interfata sau care extind clasa abstractã.

Ca si în cazul claselor abstracte, se pot defini variabile, functii sau argumente de functii de un tip interfatã. Astfel de variabile vor fi înlocuite (prin atribuire sau prin argumente efective) cu variabile de un tip clasã care implementeazã interfata respectivã. Exemplu:

```
// variantã pentru metoda Vector.toString
```

```
public final String toString() {
    StringBuffer buf = new StringBuffer();
    Enumeration e = elements();                    // variabila de tip interfata !
    buf.append("[");
    for (int i = 0 ; i <= size() ; i++) {
       String s = e.nextElement().toString();     // utilizarea variabilei interfata
       buf.append(s); buf.append(",");
    }
     buf.append("\b]");
    return buf.toString();
   }
}
```

In exemplul anterior, metoda “elements” din clasa *Vector* are ca rezultat un obiect “enumerator pe vector”, de tipul *Enumeration.* Exemplu de posibilã implementare a metodei “elements” din clasa “Vector” :

```
public Enumeration elements() {
    return new VectorEnumeration()
}
    // definirea clasei iterator pe vector
class VectorEnumeration implements Enumeration  {
        int count = 0;           // indice element curent din enumerare
        public boolean hasMoreElements() {
            return count < elementCount;       // elementCount = nr de elemente
        }
        public Object nextElement() {
             if (count < elementCount)
                 return elementData[count++];  // elementData = vector de obiecte
        }
 }
```

Clasa noastrã “VectorEnumeration” trebuie sã aibã acces la datele locale ale clasei *Vector* (“elementData” si “elementCount”), deci fie este în acelasi pachet cu clasa *Vector*, fie este o clasã interioarã clasei *Vector*.

O interfatã poate extinde o altã interfatã (*extends*) cu noi metode abstracte. Ca exemplu, interfata *TreeNode* reuneste metode de acces la un nod de arbore, iar interfata *MutableTreeNode* o extinde pe prima cu metode de modificare nod de arbore (dupã crearea unui arbore poate fi interzisã sau nu modificarea sa):

```
    public interface MutableTreeNode extends TreeNode {
         void insert(MutableTreeNode child, int index);  // adauga un fiu acestui nod
         void remove(int index);          // elimina fiu cu indice dat al acestui nod
          . . .
    }
```

Diferente între interfete si clase abstracte

O clasã poate implementa (*implements*) o interfatã sau mai multe si poate extinde (*extends*) o singurã clasã (abstractã sau nu).

Clasele care implementeazã o aceeasi interfatã pot fi foarte diverse si nu formeazã o ierarhie (o familie). Un exemplu sunt clasele cu obiecte comparabile, care toate implementeazã interfata *Comparable*: *String, Date, Integer, BigDecimal*, s.a.

O interfatã este consideratã ca un contract pe care toate clasele care implementeazã acea interfatã se obligã sã îl respecte, deci o clasã îsi asumã obligatia de a defini toate metodele din interfetele mentionate în antetul ei, putând contine si alte metode.

O clasã poate simultan sã extindã o clasã (alta decât clasa *Object*, implicit extinsã) si sã implementeze una sau mai multe interfete. Altfel spus, o clasã poate mosteni date si/sau metode de la o singurã clasã, dar poate mosteni mai multe tipuri (poate respecta simultan mai multe interfete). De exemplu, mai multe clase predefinite SDK, cu date, implementeazã simultan interfetele *Comparable* (obiectele lor pot fi comparate si la mai mic – mai mare), *Clonable* (obiectele lor pot fi copiate), *Serializable* (obiectele lor pot fi salvate sau serializate în fisiere disc sau pe alt mediu extern). Exemple:

```
public class String implements Comparable, Serializable { ...}
public class Date implements Serializable, Clonable, Comparable  { ...}
```

Este posibil ca în momentul definirii unei interfete sã nu existe nici o singurã clasã compatibilã cu interfata, cum este cazul interfetei *Comparator*.

O interfatã fãrã nici o metodã poate fi folositã pentru a permite verificarea utilizãrii unor metode numai în anumite clase, în faza de executie. Un exemplu tipic este interfata *Clonable*, definitã astfel:

```
public interface Clonable { }
```

Clasa *Object* contine metoda "clone", folositã numai de clasele care declarã cã implementeazã interfata *Clonable*. Metoda neabstractã "clone" este mostenitã automat de toate clasele Java, dar este aplicabilã numai pentru o parte din clase.

Pentru a semnala utilizarea gresitã a metodei "clone" pentru obiecte ne-clonabile, se produce o exceptie de tip *CloneNotSupportedException* atunci când ea este apelatã pentru obiecte din clase care nu aderã la interfata *Clonable*.

O utilizare asemãnãtoare o are interfata *Serializable*, pentru a distinge clasele ale cãror obiecte sunt serializabile (care contin metode de salvare si de restaurare în / din fisiere) de clasele ale cãror obiecte nu pot fi serializate (fãrã obiecte "persistente"). Practic, toate clasele cu date sunt serializabile.

Interfete ca *Serializable* si *Clonable* se numesc interfete de "marcare" a unui grup de clase ("tagging interfaces"), pentru a permite anumite verificãri.

Clasa *Object* nu contine o metodã de comparare pentru stabilirea unei relatii de precedentã între obiecte, dar dacã s-ar fi decis ca metoda "compareTo" sã facã parte din clasa *Object*, atunci ar fi fost necesarã o interfatã de "marcare" pentru a distinge clasele cu obiecte comparabile de clasele cu obiecte necomparabile.

O interfatã care stabileste un tip comun poate fi atât de generalã încât sã nu continã nici o metodã. Un exemplu este interfata *EventListener* (pachetul "java.util"),

care stabileste tipul “ascultãtor la evenimente”, dar metodele de tratare a evenimentului nu pot fi precizate nici ca prototip, deoarece depind de tipul evenimentului. Interfata este extinsã de alte interfete, specifice anumitor ascultãtori (pentru anumite evenimente):

```
public interface ActionListener extends EventListener {
   public void actionPerformed(ActionEvent e);
}
public interface ItemListener extends EventListener {
   public void itemStateChanged(ItemEvent e);
}
```

Interfetele sunt preferabile în general claselor abstracte, pentru cã oferã mai multã libertate subclaselor. Un exemplu simplu este cel al metodelor considerate esentiale pentru orice clasã dictionar: pune pereche cheie-valoare in dictionar, obtine valoarea asociatã unei chei date s.a. Aceste metode au fost reunite înainte de Java 1.2 într-o clasã abstractã (*Dictionary*) care continea numai metode abstracte. Incepând cu Java 2 aceleasi metode (plus încã câteva) au fost grupate în interfata *Map*. Avantajul solutiei interfatã în raport cu o clasã abstractã este evident atunci când definim un dictionar realizat ca un vector de perechi cheie-valoare: o clasã derivatã din *Vector* ar putea mosteni o serie de metode utile de la superclasã, dar nu poate extinde simultan clasele *Dictionary* si *Vector*.

Interfetele si clasele abstracte nu trebuie opuse, iar uneori sunt folosite împreunã într-un “framework” cum este cel al claselor colectie sau cel al claselor Swing (JFC): interfata defineste metodele ce ar trebui oferite de mai multe clase, iar clasa abstractã este o implementare partialã a interfetei, pentru a facilita definirea de noi clase prin extinderea clasei abstracte. In felul acesta se obtine o familie de clase deschisã pentru extinderi ulterioare cu clase compatibile, care nu trebuie definite însã de la zero. Un exemplu este interfata *Set* (sau *Collection*) si clasa *AbstractSet* (*AbstractCollection*) care permit definirea rapidã de noi clase pentru multimi (altele decât *HashSet* si *TreeSet*), compatibile cu interfata dar care beneficiazã de metode mostenite de la clasa abstractã.

Compararea de obiecte în Java

In unele operatii cu vectori de obiecte (sortare, determinare maxim sau minim s.a.) este necesarã compararea de obiecte (de acelasi tip). Functia de comparare depinde de tipul obiectelor comparate, dar poate fi o functie polimorficã, cu acelasi nume, tip si argumente dar cu definitii diferite. In plus, douã obiecte pot fi comparate dupã mai multe criterii (criteriile sunt variabile sau combinatii de variabile din aceste obiecte).

In Java, functia “compareTo” este destinatã comparatiei dupa criteriul cel mai “natural” (cel mai frecvent iar uneori si unicul criteriu). Pot fi comparate cu metoda “compareTo” numai clasele care implementeazã interfata *Comparable* si deci care definesc metoda abstractã “compareTo”:

```
public interface Comparable  {
```

```
  int compareTo (Object obj);          // rezultat <0 sau = 0 sau >0
}
```

Clasele care declarã implementarea acestei interfete trebuie sã continã o definitie pentru metoda "compareTo", cu argument de tip *Object*. Exemple de clase Java cu obiecte comparabile : *Integer, Float, String, Date, BigDecimal* s.a. Exemplu:

```
public class Byte extends Number implements Comparable {
  private byte value;
 . . .  //  constructori, metode specifice
  public int compareTo( Object obj) {          // compara doua obiecte Byte
    return this.value- ((Byte) obj).value ;
  }
}
```

Metoda ”compareTo” se poate aplica numai unor obiecte de tip *Comparable* si de aceea se face o conversie de la tipul general *Object* la subtipul *Comparable*. Exemplu:

```
     // determinare maxim dintr-un vector de obiecte oarecare
public static Object max (Object [ ] a) {
   Comparable maxim = (Comparable) a[0];
   for (int i=0;i<a.length;i++)
      if ( maxim.compareTo(a[i]) < 0)                // daca maxim < a[i]
         maxim=(Comparable)a[i];
   return maxim;
}
```

Functia “Arrays.sort” cu un argument foloseste implicit metoda “compareTo”.

Putem considera cã interfata *Comparable* adaugã metoda “compareTo” clasei *Object* si astfel creeazã o subclasã de obiecte comparabile.

Este posibil ca egalitatea de obiecte definitã de metoda “compareTo” sã fie diferitã de egalitatea definitã de metoda “equals”. De exemplu, clasa “Arc” produce obiecte de tip “arc de graf cu costuri”:

```
class Arc implements Comparable {
    int v,w;          // noduri capete arc
    float cost;       // cost arc
    public Arc (int x,int y) { v=x; w=y; }
    public boolean equals (Object obj) {
       Arc a= (Arc)obj;
       return (v==a.v && w==a.w);     // arce egale dacã au aceleasi extremitati
    }
    public String toString () {return v+"-"+w; }
    public int compareTo (Object obj) {
       Arc a = (Arc) obj;
       return cost – a.cost;       // arce egale dacã au costuri egale !
    }
  }
```

Metoda “equals” este folositã la cãutarea într-o colectie neordonatã (“contains”, “add” pentru multimi, s.a.), iar metoda “compareTo” este folositã la cãutarea într-o colectie ordonatã (“Collections.binarySearch”), ceea ce poate conduce la rezultate diferite pentru cele douã metode de cãutare într-o listã de obiecte “Arc”.

Pentru comparare de obiecte dupã alte criterii decât cel “natural” s-a introdus o altã functie cu numele “compare”, parte din interfata *Comparator:*

```
public interface Comparator {
   int compare (Object t1, Object t2);          // implicit abstracta, ne-statica !
   boolean equals (Object);          // comparare de obiecte comparator !
}
```

Interfata *Comparator* ar fi putut fi si o clasã abstractã cu o singurã metodã, pentru cã este improbabil ca o clasã comparator sã mai mostenească si de la o altã clasã.

Functia “Arrays.sort” are si o formã cu douã argumente; al doilea argument este de tipul *Comparator* si precizează functia de comparare (alta decât “compareTo”).

Functia care determinã maximul dintr-un vector de obiecte poate fi scrisã si astfel:

```
public static Object max (Object a[ ], Comparator c) {
   Object maxim =  a[0];
   for (int k=1; k<a.length;k++)
    if ( c.compare (maxim, a[k]) < 0)// c este un obiect comparator
       maxim =  a[k];
   return maxim;
 }
```

Functia “max” cu douã argumente este mai generalã si poate fi folositã pentru a ordona un acelasi vector dupã diferite criterii (dupã diverse proprietãti ale obiectelor). De exemplu, pentru ordonarea unui vector de siruri dupã lungimea sirurilor, vom defini urmãtoarea clasã comparator:

```
class LengthComparator implements Comparator {
  public int compare (Object t1, Object t2) {
     return ((String)t1).length() - ((String)t2).length();
  }
}
. . .  // ordonare dupa lungime
String [ ] a = {”patru”,”trei”,”unu”};
Arrays.sort( a, new LengthComparator());
```

Pentru ordonarea unei matrice de obiecte dupã valorile dintr-o coloanã datã putem defini o clasã comparator cu un constructor care primeste numãrul coloanei:

```
class CompCol implements Comparator {
 int c;                                    // numar coloana ce determina ordinea
 public CompCol (int col) { c=col; }       // constructor cu un argument
```

```
  public int compare(Object o1, Object o2) {
   Comparable c1 =(Comparable) ((Object[ ])o1)[c];      // linia o1, coloana c
   Comparable c2 =(Comparable) ((Object[ ])o2)[c];      // linia o2, coloana c
   return c1.compareTo(c2);           // compara valorile din coloana c (liniile o1 si o2)
  }
}
```

Exemplu de utilizare a acestui comparator în functia de sortare :

```
Object a[ ][ ] = { {"3","1","6"}, {"1","5","2"}, {"7","3","4"} };
for (int i=0;i<3;i++)   Arrays.sort(a,new CompCol(i));
```

Interfete pentru filtre

Un filtru este un obiect care permite selectarea (sau respingerea) anumitor obiecte dintr-o colectie sau dintr-un fisier. Un filtru poate contine o singurã metodã cu rezultat *boolean*, care sã spunã dacã obiectul primit ca argument este acceptat sau nu de filtru.
In bibliotecile Java filtrele se folosesc în clasa *File* pentru listare selectivã de fisiere dintr-un director, dar pot fi folosite si în alte situatii.

Clasa *File* din pachetul “java.io” poate genera obiecte ce corespund unor fisiere sau directoare (cataloage). Ea este destinatã operatiilor de listare sau prelucrare a fisierelor dintr-un director si nu contine functii de citire-scriere din/în fisiere.

Cel mai folosit constructor din clasa *File* primeste un argument de tip *String* ce reprezintã numele unui fisier de date sau unui fisier director. Cea mai folositã metodã a clasei *File* este metoda “list” care produce un vector de obiecte *String*, cu numele fisierelor din directorul specificat la construirea obiectului de tip *File*. Exemplu simplu de folosire a unui obiect *File* pentru afisarea continutului directorului curent:

```
import java.io.*;
class Dir {
  public static void main (String arg[]) throws IOException {
    String dir =".";                    // "numele" directorului curent
    File d =new File(dir);              // java.io.File d = new java.io.File(dir);
    String [ ] files = d.list();        // vector cu fisierele din directorul curent
    System.out.println ("Directory of "+ d.getAbsolutePath());
    for (int i=0; i< files.length;i++)
      System.out.println (files[i]);
  }
}
```

In clasa *File* existã si metode de extragere selectivã de fisiere dintr-un director:
- metoda “list” cu argument de tip *FilenameFilter* si rezultat *String*[]
- metoda “listFiles” cu argument de tip *FileFilter* sau *FilenameFilter* si rezultat *File*[]

FileFilter si *FilenameFilter* sunt interfete cu o singurã metoda “accept”.

In SDK 2 (pachetul “java.io”) sunt prevãzute douã interfete pentru clase de filtrare a continutului unui director :

```
public interface FilenameFilter {                // prezenta din jdk 1.0
   boolean accept (File path, String filename);
}
public interface FileFilter {                    // prezenta din jdk 1.2
   boolean accept (File path);
}
```

Utilizatorul are sarcina de a defini o clasã care implementeazã una din aceste interfete si de a transmite o referintã la un obiect al acestei clase fie metodei “list” fie metodei “listFiles”. Exemplu de listare selectivã cu mascã a directorului curent:

```
    // clasa pentru obiecte filtru
class FileTypeFilter implements FileFilter {
  String ext;                                    // extensie nume fisier
  public FileTypeFilter (String ext) {
    this.ext = ext;
  }
public boolean accept (File f) {
   String fname = f.getName();
    int pp = fname.indexOf('.');
    if (pp==0) return false;
      return ext.equals(fname.substring(pp+1)) ;
  }
}
// Listare fisiere de un anumit tip
class Dir {
  public static void main (String arg[ ]) throws IOException {
    File d =new File(“.”);                                  // din directorul curent
    File [ ] files = d.listFiles(new FileTypeFilter("java"));
    for (int i=0; i< files.length;i++)
        System.out.println (files[i]);
  }
}
```

Metoda “accept” definitã de utilizator este apelatã de metoda “list” (“listFiles”), care contine si ciclul în care se verificã fiecare fisier dacã satisface sau nu conditia continutã de functia “accept”. Functia “list” primeste adresa functiei “accept” prin intermediul obiectului argument de tip *FileFilter*.

Pentru filtrare dupã orice sablon (mascã) putem proceda astfel:
a) Se defineste o clasã pentru lucrul cu siruri sablon :

```
class Pattern {
  protected String mask;
  public Pattern ( String mask) {
    this.mask=mask;
```

```
  }
  public boolean match ( String str) {
    // ... compara  “str” cu “mask”
  }
}
```

b) Se defineste o clasã fltru dupã orice mascã, derivatã din clasa “Pattern” si care implementeazã o interfatã filtru din JDK :

```
class MaskFilter extends Pattern implements FileFilter {
  public MaskFilter (String mask) {
    super(mask);
  }
  public boolean accept (File f) {
    super.mask= super.mask.toUpperCase();
    String fname = f.getName().toUpperCase();
    return match(fname);
  }
}
```

c) Se foloseste un obiect din noua clasã într-o aplicatie:

```
class Dir {
  public static void main (String argv[ ]) throws IOException {
    File d =new File(argv[0]);                                  // nume director in argv[0]
    File[ ] files = d.listFiles(new MaskFilter(argv[1]));       // masca in argv[1]
    for (int i=0; i< files.length;i++)
        System.out.println (files[i]);
  }
}
```

Acest exemplu poate explica de ce *FileFilter* este o interfatã si nu o clasã abstractã: clasa filtru poate prelua prin mostenire metoda “match” de la o altã clasã, pentru cã nu este obligatã sã extindã clasa abstractã (posibilã) *FileFilter*.

Functii “callback”

O functie pentru ordonare naturalã în ordine descrescãtoare poate fi scrisã astfel:

```
public static void sortd (List list) {
    Collections.sort (a,new DComp());
}
 static class DComp implements Comparator {     // folositã intr-o metoda statica !
   public int compare (Object a,Object b) {
     Comparable ca=(Comparable)a;
     return - ca.compareTo(b);
   }
 }
```

Functia "compare" poartã numele de functie "callback". Clasa care contine functia “sortd” si clasa “DComp” transmite functiei “sort” adresa functiei “compare” (printr-un obiect “DComp”) pentru a permite functie “sort” sã se refere "înapoi" la functia de comparare. Practic, se transmite un pointer printr-un obiect ce contine o singurã functie; adresa obiectului comparator este transmisã de la “sortd” la “sort”, iar “sort” foloseste adresa continutã în obiectul comparator pentru a se referi înapoi la functia "compare" (în cadrul functiei “sort” se apeleazã functia “compare”). In limbajul C se transmite efectiv un pointer la o functie (un pointer declarat explicit).

Situatia mai poate fi schematizatã si astfel: o functie A ("main", de ex.) apeleazã o functie B ("sort", de ex.), iar B apeleazã o functie X ("compare") dintr-un grup de functii posibile. Adresa functiei X este primitã de B de la functia A. De fiecare datã când A apeleazã pe B îi transmite si adresa functiei X, care va fi apelatã înapoi de B.

La scrierea functiei de sortare (Collections.sort) nu se cunostea exact definitia functiei de comparare (pentru cã pot fi folosite diverse functii de comparare), dar s-a putut preciza prototipul functiei de comparare, ca metodã abstractã inclusã într-o interfatã. Putem deci sã scriem o functie care apeleazã functii încã nedefinite, dar care respectã toate un prototip (tip rezultat, tip si numãr de argumente).

Tehnica “callback” este folositã la scrierea unor functii de bibliotecã, dintr-un pachet de clase, functii care trebuie sã apeleze functii din programele de aplicatii, scrise ulterior de utilizatorii pachetului. Un exemplu este un analizor lexical SAX, care recunoaste marcaje dintr-un fisier XML si apeleazã metode ce vor fi scrise de utilizatori pentru interpretarea acestor marcaje (dar care nu pot fi definite la scrierea analizorului lexical). Analizorul SAX impune aplicatiei sã foloseascã functii care respectã interfetele definite de analizor si apeleazã aceste functii “callback”.

Deci functiile “callback” sunt definite de cei care dezvoltã aplicatii dar sunt apelate de cãtre functii din clase predefinite (de bibliotecã).

Clase abstracte si interfete pentru operatii de I/E

Un flux de date Java ("input/output stream") este orice sursã de date (la intrare) sau orice destinatie (la iesire), cu suport neprecizat, vãzutã ca sir de octeti. Operatiile elementare de citire si de scriere octet sunt aplicabile oricãrui tip de flux si de aceea au fost definite douã clase abstracte: o sursã de date *InputStream* si o destinatie de date *OutputStream*, care contin metodele abstracte “read” si “write”.

Ca suport concret al unui flux de date Java se considerã urmãtoarele variante: fisier disc, vector de octeti (în memorie), sir de caractere (în memorie), canal de comunicare între fire de executie (în memorie). Pentru fiecare tip de flux existã implementãri diferite ale metodelor abstracte “read” si “write” pentru citire/scriere de octeti.

Clasele abstracte *InputStream* si *OutputStream* contin si metode neabstracte pentru citire si scriere blocuri de octeti, care apeleazã metodele de citire/scriere octet, precum si alte metode care nu sunt definite dar nici nu sunt abstracte, transmise tuturor subclaselor. Exemplu:

```
public abstract class InputStream {
  public abstract int read ( ) throws IOException;          // citeste un octet
  public int read ( byte b[ ]) throws IOException {
    return read (b,0,b.length);                             // citeste un bloc de octeti
  }
  public int available( ) throws IOException {              // nr de octeti rãmasi de citit
    return 0;
  }  ... // alte metode
}
```

Clasele concrete de tip flux de citire sunt toate derivate din clasa *InputStream*, ceea ce se reflectã si în numele lor: *FileInputStream, ByteInputStream*, etc.

Metoda "read()" este redefinitã în fiecare din aceste clase si definitia ei depinde de tipul fluxului: pentru un flux fisier "read" este o metodã "nativã" (care nu este scrisã în Java si depinde de sistemul de operare gazdã), pentru un flux vector de octeti metoda "read" preia urmãtorul octet din vector.

Incepând cu versiunea 1.1 s-au introdus alte douã clase abstracte, numite *Reader* si *Writer*, cu propriile lor subclase.

```
public abstract class Reader {
    protected Reader() { ... }                          //  ptr a interzice instantierea clasei
    public int read() throws IOException {              // citire octet urmator
      char cb[ ] = new char[1];
      if (read(cb, 0, 1) == -1)
        return -1;
      else
        return cb[0];
    }
   public int read(char cbuf[ ]) throws IOException {       // citire bloc de octeti
     return read(cbuf, 0, cbuf.length);
   }
   abstract public int read(char cbuf[ ], int off, int len) throws IOException;
   abstract public void close() throws IOException;
   . . .  // alte metode
}
```

Interfata *DataInput* reuneste metode pentru citire de date de tipuri primitive:

```
public interface DataInput {
    byte readByte () throws IOException;
    char readChar () throws IOException;
    int readInt () throws IOException;
    long readLong () throws IOException;
    float readFloat () throws IOException;
    String readLine () throws IOException;
   ...
}
```

De observat cã declaratiile acestor metode nu impun modul de reprezentare externã a numerelor ci doar o formã (un prototip) pentru metodele de citire.

Clasa *RandomAccessFile* implementeazã ambele interfete si deci asigurã un fisier în care se poate scrie sau din care se poate citi orice tip primitiv. In acest tip de fisiere numerele se reprezintã în formatul lor intern (binar virgulã fixã sau virgulã mobilã) si nu se face nici o conversie la citire sau la scriere din/în fisier.

Operatiile cu fisiere depind de sistemul de operare gazdã, deci o parte din metodele claselor de I/E sunt metode "native", dar numãrul lor este mentinut la minim.

Un fragment din clasa *RandomAccessFile* aratã astfel:

```
public class RandomAccessFile implements DataOutput, DataInput {
    private native void open (String name, boolean write) throws IOException;
    public native int read () throws IOException;
    public native void write (int b) throws IOException;
    public native long length () throws IOException;
    public native void close () throws IOException;
     ... // alte metode
}
```

Clasele *DataInputStream* si *DataOutputStream* implementeazã interfetele *DataInput* si respectiv *DataOutput*. Aceste clase simplificã crearea si exploatarea de fisiere text si de fisiere binare, ce contin date de un tip primitiv (numere de diferite tipuri s.a.). Exemplu de variantã modificatã a clasei *DataInputStream* :

```
public class DataInputStream extends InputStream implements DataInput {
  InputStream in;              // “in” poate fi orice fel de flux (ca suport fizic)
  public DataInputStream (InputStream in) { this.in=in; }      // constructor
  public final byte readByte() throws IOException {
     int ch = in.read();
     if (ch < 0)  throw new EOFException();
     return (byte)(ch);
    }
   public final short readShort() throws IOException {
     InputStream in = this.in;
     int ch1 = in.read(); int ch2 = in.read();                  // citeste doi octeti
     if ((ch1 | ch2) < 0)  throw new EOFException();
     return (short)((ch1 << 8) + (ch2 << 0));          // asamblare octeti pe 16 biti
    }
  public String readLine ( ) { ... }               // citeste o linie
  public int readInt ( ) { ... }                   // citeste un numar intreg
  public float readFloat ( ) { ... }               // citeste un numar real
  . . .   // alte metode
}
```

Obiecte de tip *DataInputStream* nu pot fi create decât pe baza altor obiecte, care precizeazã suportul fizic al datelor. Clasa *DataInputStream* nu are constructor fãrã argumente si nici variabile care sã specifice suportul datelor.

Variabila publicã “in”, definitã în clasa *System*, este de un subtip (anonim) al tipului *InputStream* si corespunde tastaturii ca flux de date. Pentru obiectul adresat de variabila “System.in” se poate folosi metoda “read” pentru citirea unui octet (caracter) dar nu se poate folosi o metodã de citire a unei linii (terminate cu “Enter”).

Pentru a citi linii de la consolã, putem crea un obiect de tip *DataInputStream* , care suportã metoda “readLine”. Exemplu:

```
      // citire linie de la tastatura
public static void main (String arg[]) throws Exception {
     DataInputStream f = new DataInputStream(System.in);
     String line= f.readLine();
     System.out.println (“ S-a citit: "+ line);
 }
```

Utilizarea metodelor “readInt”, “readFloat” s.a. nu are sens pentru tastaturã, deoarece trebuie fãcutã conversia de la sir de caractere la întreg (format intern binar), operatie care nu se face în aceste metode. Dacã o linie contine un singur numãr care nu este neapãrat întreg, atunci putem scrie:

```
    DataInputStream f = new DataInputStream(System.in);
     String line= f.readLine();
    double d = Double.parseDouble(line);
```

Variabila publicã “out” din clasa *System* nu este de un subtip al tipului *OutputStream*, ci este de un subtip (anonim) al tipului *PrintStream*, tip care contine mai multe metode “print”, care fac conversia numerelor din format intern binar în sir de caractere, înainte de scriere.

7. Colectii de obiecte în Java 2

Infrastructura claselor colectie

O colectie este un obiect ce contine un numãr oarecare, variabil de obiecte. Colectiile se folosesc pentru memorarea si regãsirea unor date sau pentru transmiterea unui grup de date de la o metodã la alta. Colectiile Java sunt structuri de date generice, realizate fie cu elemente de tip *Object*, fie cu clase sablon (cu tipuri parametrizate) .

Infrastructura colectiilor (Collection Framework) oferã clase direct utilizabile si suport pentru definirea de noi clase (sub formã de clase abstracte), toate conforme cu anumite interfete ce reprezintã tipuri abstracte de date (liste, multimi, dictionare).

Un utilizator îsi poate defini propriile clase colectie, care respectã aceste interfete impuse si sunt compatibile cu cele existente (pot fi înlocuite unele prin altele).

Clasele abstracte existente fac uz de iteratori si aratã cã aproape toate metodele unei clase colectie pot fi scrise numai folosind obiecte iterator (cu exceptia metodelor de adãugare obiect la colectie si de calcul numar elemente din colectie, care depind de structura fizicã a colectiei).

Familia claselor colectie din Java este compusã din douã ierarhii de clase :

- Ierarhia care are la bazã interfata *Collection*,
- Ierarhia care are la bazã interfata *Map*.

O colectie, în sensul Java, este un tip de date abstract care reuneste un grup de obiecte de tip *Object*, numite si elemente ale colectiei. Un dictionar (o asociere) este o colectie de perechi chei-valoare (ambele de tip *Object*); fiecare pereche trebuie sã aibã o cheie unicã, deci nu pot fi douã perechi identice.

Interfata *Collection* contine metode aplicabile pentru orice colectie de obiecte. Nu toate aceste operatii trebuie implementate obligatoriu de clasele care implementeazã interfata *Collection*; o clasã care nu defineste o metodã optionalã poate semnala o exceptie la încercarea de apelare a unei metode neimplementate.

Urmeazã descrierea completã a interfetei *Collection* :

```
public interface Collection {                  // operatii generale ptr orice colectie
  int size();                                  // dimensiune colectie (nr de elemente)
  boolean isEmpty();                           // verifica daca colectie vida
  boolean contains(Object element);     // daca colectia contine un obiect dat
  boolean add(Object element);                 // adauga un element la colectie
  boolean remove(Object element);              // elimina un element din colectie
  Iterator iterator();                         // produce un iterator pentru colectie
  boolean containsAll(Collection c);           // daca colectia contine elem. din colectia c
  boolean addAll(Collection c);                // adauga elem. din c la colectie
  boolean removeAll(Collection c);             //elimina din colectie elem. colectiei c
  boolean retainAll(Collection c);             // retine in colectie numai elem. din c
  void clear();                                // sterge continut colectie
  Object[ ] toArray();                         // copiere colectie intr-un vector
}
```

Din interfata *Collection* sunt derivate direct douã interfete pentru tipurile abstracte:

- *Set* pentru multimi de elemente distincte.
- *List* pentru secvente de elemente, în care fiecare element are un succesor si un predecesor si este localizabil prin pozitia sa în listã (un indice întreg).

Pentru fiecare din cele 3 structuri de date abstracte (listã, multime,dictionar) sunt prevãzute câte douã clase concrete care implementeazã interfetele respective. Structurile de date concrete folosite pentru implementarea tipurilor de date abstracte sunt : vector extensibil, listã dublu înlãntuitã, tabel de dispersie si arbore binar.

Toate clasele colectie instantiabile redefinesc metoda “toString”, care produce un sir cu toate elementele colectiei, separate prin virgule si încadrate de paranteze drepte. Afisarea continutului unei colectii se poate face printr-o singurã instructiune.

De asemenea sunt prevãzute metode de trecere de la vectori intrinseci de obiecte (*Object* []) la colectii de obiecte si invers: functia “Arrays.asList”, cu un argument vector intrinsec de obiecte si rezultat de tip *List* si functia “toArray” din clasa *AbstractCollection*, de tip *Object*[]. Exemplu:

```
String sv[] = {“unu”,”doi”,”trei”};
List list = Arrays.asList(sv);     //  nu este nici ArrayList, nici LinkedList !
System.out.println (list);         // System.out.println (list.toString());
String aux[] = (String[ ]) list.toArray();    // aux identic cu sv
```

O a treia ierarhie are la bazã interfata *Iterator*, pentru metodele specifice oricãrui iterator asociat unei colectii sau unui dictionar. Toate colectiile au iteratori dar nu si clasele dictionar (se poate însã itera pe multimea cheilor sau pe colectia de valori).

Clasa *Collections* contine metode statice pentru mai multi algoritmi "generici" aplicabili oricãrei colectii: "min", "max" (valori extreme dintr-o colectie). Alte metode sunt aplicabile numai listelor: “sort”, “binarySearch”, “reverse”, “shuffle”. Algoritmii generici se bazeazã pe existenta tipului generic *Object* si a metodelor polimorfice (equals, compareTo s.a.)

Multimi de obiecte

O multime este o colectie de elemente distincte pentru care operatia de cãutare a unui obiect în multime este frecventã si trebuie sã aibã un timp cât mai scurt.

Interfata *Set* contine exact aceleasi metode ca si interfata *Collection*, dar implementãrile acestei interfete asigurã unicitatea elementelor unei multimi. Metoda de adãugare a unui obiect la o multime "add" verificã dacã nu existã deja un element identic, pentru a nu se introduce duplicate în multime. De aceea obiectele introduse în multime trebuie sã aibã metoda “equals” redefinitã, pentru ca obiecte diferite sã nu aparã ca fiind egale la compararea cu “equals”.

Clasele concrete API care implementeazã interfata *Set* sunt:
HashSet : pentru o multime neordonatã realizatã ca tabel de dispersie
T*reeSet* : pentru o multime ordonatã realizatã ca un arbore echilibrat

Tabelul de dispersie asigurã cel mai bun timp de cãutare, dar arborele echilibrat permite pãstrarea unei relatii de ordine între elemente.

Exemplul următor foloseste o multime de tipul *HashSet* pentru un graf reprezentat prin multimea arcelor (muchiilor) :

```
  //  graf reprezentat prin multimea arcelor
class Graph {
  private int n;                          // numar de noduri in graf
  private Set arc;                        // lista de arce
  public Graph ( int n) {
    this.n =n;
    arc = new HashSet();
  }
  public void addArc (int v,int w) {      // adauga arcul (v,w) la graf
    arc.add (new Arc (v,w));
  }
  public boolean isArc (int v,int w) {    // daca exista arcul (v,w)
    return arc.contains (new Arc(v,w));
  }
  public String toString () {
     return arc.toString();
  }
}
```

In general se recomandă programarea la nivel de interfată si nu la nivel de clasă concretă. Asadar, se vor folosi pe cât posibil variabile de tip *Collection* sau *Set* si nu variabile de tip *HashSet* sau *TreeSet* (numai la construirea obiectului colectie se specifică implementarea sa).

Operatiile cu două colectii sunt utile mai ales pentru operatii cu multimi:

```
s1.containsAll (s2)     // true dacă s1 contine pe s1 (includere de multimi)
s1.addAll (s2)          // reuniunea multimilor s1 si s2 (s1=s1+s2)
s1.retainAll (s2)       // intersectia multimilor s1 si s2 (s1=s1*s2)
s1.removeAll (s2)       // diferenta de multimi (s1= s1-s2)
```

Diferenta simetrică a două multimi se poate obtine prin secventa următoare:

```
Set sdif = new HashSet(s1);
sdif.addAll(s2);                        // reuniune s1+s2
Set aux = new HashSet(s1);
aux.retainAll (s2);                     // intersectie s1*s2 in aux
simdif.removeAll(aux);                  // reuniune minus intersectie
```

Exemplul următor aratã cum putem folosi două multimi pentru afisarea cuvintelor care apar de mai multe ori si celor care apar o singură dată într-un text. S-au folosit multimi arbori ordonati, pentru a permite afisarea cuvintelor în ordine lexicografică.

```
class Sets {
  public static void main (String arg[ ]) throws IOException {
    Set toate = new HashSet ();                 // toate cuvintele distincte din text
```

```
    Set dupl =new TreeSet ();                    // cuvintele cu aparitii multiple
    RandomAccessFile f = new RandomAccessFile (arg[0],"r");
    String sir="", line;
    while ( (line =f.readLine ()) != null)       // citeste tot fisierul in sir
      sir=sir+line+"\n";
    StringTokenizer st = new StringTokenizer (sir);
    while ( st.hasMoreTokens() ) {
      String word= st.nextToken();
      if ( ! toate.add (word) )                  // daca a mai fost in "toate"
        dupl.add (word);                         // este o aparitie multipla
    }
   System.out.println ("multiple: "+ dupl);
   Set unice = new TreeSet (toate);
   unice.removeAll (dupl);                       // elimina cuvinte multiple
   System.out.println ("unice: " + unice);
  }
}
```

Interfata *SortedSet* extinde interfata *Set* cu câteva metode aplicabile numai pentru colectii ordonate:

Object first(), Object last(), SortedSet subSet(Object from, Object to),
SortedSet headSet(Object to), SortedSet tailSet (Object from).

Liste secventiale

Interfata *List* contine câteva metode suplimentare fatã de interfata *Collection* :

- Metode pentru acces pozitional, pe baza unui indice întreg care reprezintã pozitia :
  get (index), set (index,object), add (index, object), remove (index)
- Metode pentru determinarea pozitiei unde se aflã un element dat, deci cãutarea primei si ultimei aparitii a unui obiect într-o listã:
  indexOf (object), lastIndexOf (object)
- Metode pentru crearea de iteratori specifici listelor:
  listIterator (), listIterator (index)
- Metodã de extragere sublistã dintr-o listã:
  List subList(int from, int to);

Existã douã implementãri pentru interfata *List*:
ArrayList listã vector, preferabilã pentru acces aleator frecvent la elementele listei.
LinkedList listã dublu înlãntuitã, preferatã când sunt multe inserãri sau stergeri.

In plus, clasei *Vector* i-au fost adãugate noi metode pentru a o face compatibila cu interfata *List*. Noile metode au nume mai scurte si o altã ordine a argumentelor în metodele "add" si "set":

Forma veche (1.1)	Forma nouã (1.2)
Object elementAt (int)	Object get(int)
Object setElementAt (Objext, int)	Object set (i, Object)
void insertElementAt (Object, int)	void add (i,Object)

Exemplu de functie care schimbã între ele valorile a douã elemente i si j:

```
static void swap (List a, int i, int j) {          // din clasa Collections
  Object aux = a.get(i);                           // a.elementAt(i)
  a.set (i,a.get(j));                              // a.setElementAt (a.elementAt(j) , i)
  a.set (j,aux);                                   // a.setElementAt (aux , j)
}
```

De observat cã metodele "set" si "remove" au ca rezultat vechiul obiect din listã, care a fost modificat sau eliminat. Metoda “set” se poate folosi numai pentru modificarea unor elemente existente în listã, nu si pentru adãugare de noi elemente.

Algoritmii de ordonare si de cãutare binarã sunt exemple de algoritmi care necesitã accesul direct, prin indici, la elementele listei. Urmeazã o variantã simplificatã a metodei “binarySearch”; în caz de obiect negãsit, rezultatul este pozitia unde ar trebui inserat obiectul cãutat astfel ca lista sã rãmânã ordonatã (decalatã cu 1 si cu minus).

```
public static int binSearch (List list, Object key) {
    int low = 0, high = list.size()-1;
    while (low <= high) {
        int mid =(low + high)/2;
        Object midVal = list.get(mid);                    // acces prin indice
        int cmp = ((Comparable)midVal).compareTo(key);
        if (cmp < 0) low = mid + 1;
        else
            if (cmp > 0) high = mid - 1;
            else return mid;                              // gasit
     }
     return -(low + 1);                                   // negasit
}
```

Exemplu de utilizare a functiei de cãutare într-o secventã care adaugã un obiect la o listã ordonatã, astfel ca lista sã rãmânã ordonatã:

```
int pos = Collections.binarySearch (alist,akey);
if (pos < 0)
    alist.add (-pos-1, akey);
```

Accesul pozitional este un acces direct, rapid la vectori dar mai putin eficient în cazul listelor înlãntuite. De aceea s-a definit o clasã abstractã *AbstractSequentialList*, care este extinsã de clasa *LinkedList* dar nu si de clasa *ArrayList*. Metoda staticã Collections.binarySearch, cu parametru de tipul general *List*, recunoaste tipul de listã si face o cãutare binarã în vectori, dar o cãutare secventialã în liste secventiale.

Clasa *LinkedList* contine, în plus fatã de clasa abstractã *AbstractList*, urmãtoarele metode, utile pentru cazuri particulare de liste (stive, cozi etc.): getFirst(), getLast(), removeFirst(), removeLast(), addFirst(Object), addLast(Object).

Clase dictionar

Interfata *Map* contine metode specifice operatiilor cu un dictionar de perechi cheie valoare, în care cheile sunt unice . Existã trei implementãri pentru interfata *Map*:
HashMap dictionar realizat ca tabel de dispersie, cu cel mai bun timp de cãutare.
TreeMap dictionar realizat ca arbore echilibrat, care garanteazã ordinea de enumerare.
LinkedHashMap tabel de dispersie cu mentinere ordine de introducere (din vers. 1.4)
Definitia simplificatã a interfetei *Map* este urmãtoarea:

```
public interface Map {
  Object put(Object key, Object value);      // pune o pereche cheie-valoare
  Object get(Object key);                    // extrage valoare asociatã unei chei date
  Object remove(Object key);                 // eliminã pereche cu cheie datã
  boolean containsKey(Object key);           // verifica daca exista o cheie data
  boolean containsValue(Object value);// verifica daca exista o valoare data
  int size();                                // dimensiune dictionar (nr de perechi)
  boolean isEmpty();
  void clear();                              // elimina toate perechile din dictionar
  public Set keySet();                       // extrage multimea cheilor
  public Collection values();                // extrage valori din dictionar
}
```

Metoda “get” are ca rezultat valoarea asociatã unei chei date sau *null* dacã cheia datã nu se aflã în dictionar. Metoda “put” adaugã sau modificã o pereche cheie-valoare si are ca rezultat valoarea asociatã anterior cheii date (perechea exista deja în dictionar) sau *null* dacã cheia perechii introduse este nouã. Efectul metodei “put (k,v)” în cazul cã existã o pereche cu cheia “k” în dictionar este acela de înlocuire a valorii asociate cheii “k” prin noua valoare “v” (valoarea înlocuitã este transmisã ca rezultat al metodei “put”).

Testul de apartenentã a unei chei date la un dictionar se poate face fie direct prin metoda “containsKey”, fie indirect prin verificarea rezultatului operatiei “get”.

Clasele care generează obiecte memorate într-un obiect *HashMap* sau *HashSet* trebuie sã redefineascã metodele "equals" si "hashCode", astfel încât sã se poatã face cãutarea la egalitate dupã codul de dispersie.

In exemplul urmãtor se afiseazã numãrul de aparitii al fiecãrui cuvânt distinct dintr-un text, folosind un dictionar-arbore pentru afisarea cuvintelor în ordine.

```
class FrecApar {                    // frecventa de aparitie a cuvintelor intr-un text
  private static final Integer ONE= new Integer(1);        // o constanta
  public static void main (String arg[]) {
    Map dic = new TreeMap ();
    String text =“ trei unu doi trei doi trei “;
    StringTokenizer st = new StringTokenizer (new String (text));
    String word;
    while ( st.hasMoreTokens()) {
      word = st.nextToken();
      Integer nrap = (Integer) dic.get(word);
      if (nrap == null)
        dic.put (word,ONE);                                  // prima aparitie
```

```
     else
        dic.put (word, new Integer (nrap.intValue()+1));    // alta aparitie
    }
    System.out.println (dic);                               // afisare dictionar
 }
}
```

Cheile dintr-un dictionar pot fi extrase într-o multime cu metoda “keySet”, iar valorile din dictionar pot fi extrase într-o colectie (o listã) cu metoda “values”.Metoda “entrySet” produce o multime echivalentã de perechi "cheie-valoare", unde clasa pereche are tipul *Entry*. *Entry* este o interfatã inclusã în interfata *Map* si care are trei metode: “getKey”, “getValue” si “setValue”.

De observat cã metodele “entrySet”,“keySet” si “values” (definite în *AbstractMap*) creeazã doar imagini noi asupra unui dictionar si nu alte colectii de obiecte; orice modificare în dictionar se va reflecta automat în aceste “imagini”, fãrã ca sã apelãm din nou metodele respective. O imagine este creatã printr-o clasã care defineste un iterator pe datele clasei dictionar si nu are propriile sale date. Metodele clasei imagine sunt definite apoi pe baza iteratorului, care dã acces la datele din dictionar.

Diferenta dintre clasele *HashMap* si *LinkedHashMap* apare numai în sirul produs de metoda “toString” a clasei: la *LinkedHashMap* ordinea perechilor în acest sir este aceeasi cu ordinea de introducere a lor în dictionar, în timp ce la *HashMap* ordinea este aparent întâmplãtoare (ea depinde de capacitatea tabelei de dispersie, de functia “hashCode” si de ordinea de introducere a cheilor). Pentru pãstrarea ordinii de adãugare se foloseste o listã înlãntuitã, iar tabelul de dispersie asigurã un timp bun de regãsire dupã cheie.

Colectii ordonate

Problema ordonãrii este rezolvatã diferit pentru liste fatã de multimi si dictionare. Listele sunt implicit neordonate (se adaugã numai la sfârsit de listã) si pot fi ordonate numai la cerere, prin apelul unei metode statice (Collections.sort). Multimile, ca si dictionarele, au o variantã de implementare (printr-un arbore binar ordonat) care asigurã mentinerea lor în ordine dupã orice adãugare sau eliminare de obiecte.

Exemplu de ordonare a unei liste:

```
public static void main (String arg[ ]) {
  String tab[ ] = {"unu","doi","trei",”patru”,”cinci”};
  List lista = Arrays.asList (tab);
  Collections.sort (lista);
  System.out.println (lista);  // [cinci,doi,patru,trei,unu]
}
```

Putem sã ne definim vectori sau liste ordonate automat, sau alte alte structuri compatibile cu interfata *List* si care asigurã ordinea (un arbore binar, de exemplu).

Exemplu de încercare de a defini o clasã pentru o multime ordonatã implementatã printr-un vector (dupã modelul clasei *TreeSet*):

```
public class SortedArray extends ArrayList implements List {
  Comparator cmp=null;                          // adresa obiectului comparator
  public SortedArray () { super();}
  public SortedArray (Comparator comp) {
    super();    cmp=comp;                       // retine adresa obiectului comparator
  }
  public boolean add (Object obj) {       // adaugare la vector ordonat
    boolean modif= super.add(obj);        // daca s-a modificat colectia dupa adaugare
    if ( ! modif) return modif;           // colectia nemodificata ramane ordonata
    if ( cmp==null)
      Collections.sort (this);            // ordonare dupa criteriul natural
    else
      Collections.sort (this,cmp);        // ordonare cu comparator primit din afara
    return modif;
  }
}
```

Problema clasei anterioare este cã ar trebui redefinitã si metoda “set” care modificã elemente din listã, pentru a mentine lista ordonatã. Acest lucru nu este posibil prin folosirea metodei “Collections.sort” deoarece aceasta apeleazã metoda “set” si apare o recursivitate indirectã infinitã de forma set -> sort -> set -> sort -> set ->…

O multime ordonatã este de tipul *TreeSet* iar un dictionar ordonat este de tipul *TreeMap*. Se pot defini si alte tipuri de colectii sau dictionare ordonate, care implementeazã (optional) interfata *SortedSet*, respectiv interfata *SortedMap*. Adãugarea sau modificarea valorilor dintr-un arbore se face cu mentinerea ordinii si nu necesitã reordonarea multimii sau dictionarului. Obiectele introduse într-o colectie *TreeSet* sau *TreeMap* trebuie sã apartinã unei clase care implementeazã interfata *Comparable* si contine o definitie pentru metoda “compareTo”.

Exemplu de ordonare a unei liste de nume (distincte) prin crearea si afisarea unei multimi ordonate:

```
SortedSet lst = new TreeSet (lista);      // sau se adauga cu metoda addAll
```

Iteratorul unei colectii ordonate parcurge elementele în ordinea dictatã de obiectul comparator folosit mentinerea colectiei în ordine.

O problemã comunã colectiilor ordonate este criteriul dupã care se face ordonarea, deci functia de comparatie, care depinde de tipul obiectelor comparate. Sunt prevãzute douã solutii pentru aceastã problemã, realizate ca douã interfete diferite si care au aplicabilitate distinctã.

Anumite metode statice (sort, min, max s.a.) si unele metode din clase pentru multimi ordonate apeleazã în mod implicit metoda "compareTo", parte a interfetei *Comparable*. Clasele JDK cu date (*String, Integer, Date* s.a.) implementeazã interfata *Comparable* si deci contin o metoda "compareTo" pentru o ordonare "naturalã" (exceptie face clasa *Boolean*, ale cãrei obiecte nu sunt comparabile).

Ordinea naturalã este ordinea valorilor algebrice (cu semn) pentru toate clasele numerice, este ordinea numericã fãrã semn pentru caractere si este ordinea lexicograficã pentru obiecte de tip *String*. Pentru alte clase, definite de utilizatori, trebuie implementatã interfata *Comparable* prin definirea metodei "compareTo", dacã obiectele clasei pot fi comparate (si sortate).

Pentru ordonarea dupã un alt criteriu decât cel natural si pentru ordonarea dupã mai mlte criterii se va folosi o clasã compatibilã cu interfata *Comparator*.

Rezultatul metodei "compare" este acelasi cu al metodei "compareTo", deci un numãr negativ dacã ob1<ob2, zero dacã ob1==ob2 si un numãr pozitiv dacã ob1>ob2.

Un argument de tip *Comparator* apare în constructorii unor clase si în câteva metode din clasa *Collections* (sort, min, max) ce necesitã compararea de obiecte.

Pentru a utiliza o functie “compare” trebuie definitã o clasã care contine numai metoda "compare", iar un obiect al acestei clase se transmite ca argument functiei. Exemplu de ordonare a dictionarului de cuvinte-frecventã creat anterior, în ordinea inversã a numãrului de aparitii:

```
Set entset = dic.entrySet();
ArrayList entries = new ArrayList(entset);
Collections.sort (entries, new Comp());
. . .
// clasa comparator obiecte Integer in ordine inversa celei naturale
class Comp implements Comparator {
 public int compare (Object o1, Object o2) {
  Map.Entry  e1= (Map.Entry)o1, e2= (Map.Entry)o2;   // e1,e2=prechi cheie-val
  return ((Integer)e2.getValue()).compareTo ((Integer) e1.getValue());
 }
}
```

Clase iterator

Una din operatiile frecvente asupra colectiilor de date este enumerarea tuturor elementelor colectiei (sau a unei subcolectii) în vederea aplicãrii unei prelucrãri fiecãrui element obtinut prin enumerare. Realizarea concretã a enumerãrii depinde de tipul colectiei si foloseste un cursor care înaintează de la un element la altul, într-o anumitã ordine (pentru colectii neliniare). Cursorul este un indice întreg în cazul unui vector sau un pointer (o referintã) pentru o listã înlãntuitã sau pentru un arbore binar. Pentru colectii mai complexe, cum ar fi vectori de liste, enumerarea poate folosi indici (pentru vectori) si pointeri (pentru noduri legate prin pointeri).

Generalizarea modului de enumerare a elementelor unei colectii pentru orice fel de colectie a condus la aparitia claselor cu rol de “iterator” fatã de o altã clasã colectie. Orice clasã colectie Java 2 poate avea o clasã iterator asociatã. Pentru un acelasi obiect colectie (de ex. un vector) pot exista mai multi iteratori, care progresează în mod diferit în cadrul colectiei, pentru cã fiecare obiect iterator are o variabilã cursor proprie.

Toate clasele iterator rebuie sã includã urmãtoarele operatii:

- Pozitionarea pe primul element din colectie

- Pozitionarea pe urmãtorul element din colectie
- Obtinerea elementului curent din colectie
- Detectarea sfârsitului colectiei (test de terminare a enumerãrii).

Interfata *Iterator* contine metode generale pentru orice iterator:

```
public interface Iterator {
   boolean hasNext();              // daca exista un element urmator in colectie
   Object next();                  // extrage element curent si avans la urmãtorul
   void remove();                  // elimina element curent din colectie (optional)
}
```

De observat cã modificarea continutului unei colectii se poate face fie prin metode ale clasei colectie, fie prin metoda “remove” a clasei iterator, dar nu ar trebui folosite simultan ambele moduri de modificare. In exemplul urmãtor apare o exceptie de tip *ConcurrentModificationException*:

```
ArrayList a = new ArrayList();
 Iterator it = a.iterator();
while (it.hasNext()) {
  it.next(); it.remove(); a.add( "x");
 }
```

Pentru fiecare clasã concretã de tip colectie existã o clasã iterator. Un obiect iterator este singura posibilitate de enumerare a elementelor unei multimi si o alternativã pentru adresarea prin indici a elementelor unei liste. Un dictionar nu poate avea un iterator, dar multimea perechilor si multimea cheilor din dictionar au iteratori.

Clasele iterator nu sunt direct instantiabile (nu au constructor public), iar obiectele iterator se obtin prin apelarea unei metode a clasei colectie (metoda “iterator”). In felul acesta, programatorul este obligat sã creeze întâi obiectul colectie si numai dupã aceea obiectul iterator. Mai multi algoritmi generici realizati ca metode statice (în clasa *Collections*) sau ca metode ne-statice din clasele abstracte folosesc un obiect iterator pentru parcurgerea colectiei. Exemplu:

```
public static void sort (List list) {
    Object a[] = list.toArray();      // transforma lista in vector intrinsec
    Arrays.sort(a);                   // ordonare vector intrinsec (mai eficienta)
    ListIterator i = list.listIterator();
    for (int j=0; j<a.length; j++) {  // modificare elemente din lista
       i.next();  i.set(a[j]);
    }
}
```

Fragmentul urmãtor din clasa *AbstractCollection* aratã cum se pot implementa metodele unei clase colectie folosind un iterator pentru clasa respectivã:

```
public abstract class AbstractCollection implements Collection {
 public boolean contains(Object o) {          //  fara cazul o==null
```

```
    Iterator e = iterator();
    while (e.hasNext())
      if (o.equals(e.next()))
            return true;
    return false;
 }
 public Object[] toArray() {
    Object[] result = new Object[size()];
    Iterator e = iterator();
    for (int i=0; e.hasNext(); i++)
      result[i] = e.next();
    return result;
 }
 public boolean containsAll(Collection c) {
    Iterator e = c.iterator();
    while (e.hasNext())
      if(!contains(e.next()))
          return false;
    return true;
 }
 public boolean addAll (Collection c) {
    boolean modified = false;
    Iterator e = c.iterator();
    while (e.hasNext()) {
      if(add(e.next()))
         modified = true;
    }
    return modified;
 }
  . . .  // alte metode
}
```

Interfata *ListIterator* contine metode pentru traversarea unei liste în ambele sensuri si pentru modificarea elementelor enumerate : hasNext(), next(), hasPrevious(), previous(), nextIndex(), previousIndex(), remove(), set (Object o), add(Object o).

Exemplu de parcurgere a unei liste de la coadã la capãt:

```
List a = new LinkedList();
for (ListIterator i= a.listIterator (a.size()); i.hasPrevious(); )
   System.out.println (i.previous());
```

Metoda “listIterator()” pozitioneazã cursorul pe începutul listei, iar metoda “listIterator(index)” pozitioneazã initial cursorul pe indicele specificat.

O clasã dictionar (Map) nu are un iterator asociat, dar este posibilã extragerea unei multimi de chei sau unei multimi de perechi cheie-valoare dintr-un dictionar, iar aceste multimi pot fi enumerate.

Secventa urmãtoare face o enumerare a perechilor cheie-valoare dintr-un dictionar “map”, folosind interfata publicã *Entry*, definitã în interiorul interfetei *Map*:

```
for (Iterator it= map.entrySet().iterator(); it.hasNext();) {
    Map.Entry e = (Map.Entry) it.next();
    System.out.println ( e.getKey()+":"+e.getValue());
}
```

Definirea de noi clase colectie

Definirea unor noi clase colectie poate fi necesarã din mai multe motive:
- Cerinte de performantã (de memorie ocupatã sau timp de acces) impun alte implementãri pentru tipurile abstracte de date existente; de exemplu o multime de obiecte realizatã ca vector sau ca listã simplu înlãntuitã.
- Sunt necesare alte tipuri abstracte de date neimplementate în JDK 1.2 : graf, coadã simplã, coadã cu prioritãti, colectie de multimi disjuncte, arbore binar s.a.

Aceste noi clase colectie ar trebui sã se conformeze cadrului creat de interfetele *Collection, Set, List, Map, Iterator* s.a. si pot sã refoloseascã metode definite în clasele abstracte si în clasele instantiabile deja existente.

O coadã este un alt caz particular de listã si poate fi derivatã dintr-o listã astfel:

```
public class Queue extends LinkedList {
  public boolean add (Object obj) {          // adãugare obiect la o coadã
    addFirst(obj); return true;
  }
  public Object remove () {                  // extragere obiect din coadã
    return removeLast ();
  }
}
```

Lista înlãntuitã din clasa *LinkedList* este o listã circularã cu santinelã, iar accesul la ultimul element din listã se face direct si rapid, deoarece el se aflã imediat înaintea elementului santinelã.

Clasa urmãtoare defineste o multime realizatã ca listã înlãntuitã si mosteneste aproape toate operatiile si mecanismul iterator de la clasa *LinkedList*:

```
public class LinkSet extends LinkedList implements Set {
  public boolean add (Object obj) {
    // adauga la sfirsit daca nu exista deja
    if ( contains(obj))
      return false;
    return super.add(obj);
  }
}
```

In Java, elementele unei colectii pot fi de orice tip derivat din *Object*, deci pot fi la rândul lor colectii, ceea ce simplificã definirea si utilizarea colectiilor de colectii. Exemplul urmãtor este o clasã pentru grafuri memorate prin liste de adiacente. Se foloseste un vector de liste înlãntuite:

```
public class Graph {                    // graf reprezentat prin liste de adiacente
  private int n;                        // numar de noduri in graf
  private List a;                       // vector sau lista de noduri
  public Graph ( int n) {
    this.n =n; a = new ArrayList (n);   // aloca memorie pentru vector
    for (int v=0;v<n;v++)
      a.add (new LinkedList());         // adauga o lista vida de vecini
  }
  public void addArc (int v,int w) {    // adauga arcul (v,w) la graf
    List vlist = (List)a.get(v);        // lista de vecini ai nodului v
    vlist.add(new Integer(w));          // adauga w la vecinii lui v
  }
  public boolean isArc (int v,int w) {          // verifica daca arc de la v la w
    List vlist = (List)a.get(v);                // lista vecinilor lui v
    return vlist.contains (new Integer(w));     // daca contine nodul w
  }
  public String toString () {                   // un sir cu toate listele de adiac.
     return a.toString();
  }
}
```

Clasa abstractã *AbstractCollection*, compatibilã cu interfata *Collection*, contine implementãri pentru o serie de operatii care pot fi realizate indiferent de tipul colectiei, folosind iteratori : isEmpty, contains, toArray, remove, containsAll, addAll, removeAll, retainAll, clear, toString.

Clasele abstracte *AbstractSet* si *AbstractList* extind clasa *AbstractCollection*, iar clasele instantiabile pentru multimi si liste extind aceste clase abstracte. In acest fel se mostenesc toate functiile mentionate, desi unele din ele sunt redefinite din motive de performantã.

Prin extinderea claselor abstracte se pot defini noi clase colectie cu un efort de programare minim si cu pãstrarea compatibilitãtii cu interfetele comune (*List, Set, Map*). Exemplu de clasã pentru obiecte dictionar realizate ca vectori de perechi:

```
public class ArrayMap extends AbstractMap {     // dictionar vector de perechi
  private ArrayList entries ;                    // vector de perechi cheie-valoare
  public ArrayMap (int n) { entries = new ArrayList(n); }
  public Object put ( Object key, Object value) {
    int i = entries.indexOf( new MEntry(key,value));
    if (i<0) {          // daca cheia key nu exista anterior in dictionar
      entries.add (new MEntry(key,value));  // se adauga o noua pereche
      return null;
    }
    else {              // daca cheia key exista deja in dictionar
      Object v = ((MEntry) entries.get(i)).getValue();   // valoare asociata anterior cheii
      entries.set (i, new MEntry(key,value));            // modifica perechea
      return v;
    }
  }
  public Set entrySet ( ) {                      // multimea cheilor
```

```
    return new HashSet(entries);
  }
  public String toString( ) {
    return entries.toString();
  }
}
```

Metoda “toString” a fost redefinitã numai pentru a obtine un sir în care se pãstreazã ordinea de adãugare la dictionarul vector. De remarcat cã metoda “entrySet” nu putea folosi o multime *TreeSet* decât dacã clasa “MEntry” implementeazã interfata *Comparable*.

De fapt, metoda “entrySet” trebuie scrisã altfel, fãrã crearea unei noi multimi cu date care existã deja în dictionar; ea trebuie sã ofere doar o imagine de multime asupra vectorului “entries” continut în dictionar (asa cum se face si în clasele JDK):

```
  public Set entrySet () {
    return new ArraySet();       // o clasa fara date proprii !
  }
  class ArraySet extends AbstractSet {
    public int size() { return entries.size(); }
    public Iterator iterator () { return new ASIterator();}
  }
  // clasa iterator (inclusa in clasa ArrayMap)
  class ASIterator implements Iterator {
    int i;
    ASIterator () { i=0; }
    public boolean hasNext() { return i < entries.size();}
    public Object next() {
      Object e=  (MEntry)entries.get(i);
      i++; return e;
    }
    public void remove () {
      entries.remove(i-1);
    }
  }
```

Clasa “MEntry” implementeazã interfata “Map.Entry” si redefineste metoda “equals” (eventual si metoda “toString”):

```
  class MEntry implements Map.Entry {
     private Object key,val;
     public MEntry (Object k, Object v) {
       key=k; val=v;
     }
     public Object getKey() { return key; }
     public Object getValue() { return val;}
     public Object setValue (Object v) { val=v; return v;}
     public boolean equals (Object obj) {
       return ((MEntry)obj).getKey().equals(key);
```

```
    }
    public String toString() { return key+":"+val;}
 }
```

Clase colectie "sablon"

Clasele colectie cu tipuri parametrizate au fost introduse în versiunea 1.5, ca solutii alternative pentru colectiile de obiecte deja existente. Exemple de declarare a unor vectori cu elemente de diferite tipuri :

```
ArrayList<Integer> a = new ArrayList<Integer>();
ArrayList<String> b = ArrayList<String> (100);
ArrayList<TNode> c = ArrayList<TNode> (n);
ArrayList<LinkedList<Integer>> graf;                    // initializata ulterior
```

Avantajele sunt acelea cã se poate verifica la compilare dacã se introduc în colectie numai obiecte de tipul declarat pentru elementele colectiei, iar la extragerea din colectie nu mai este necesarã o conversie de la tipul generic la tipul specific aplicatiei.
Exemplu cu un vector de obiecte *Integer*:

```
ArrayList<Integer> list = new ArrayList<Integer>();
for (int i=1;i<10;i++)
  list.add( new Integer(i) );          // nu se poate adauga alt tip decat Integer
int sum=0;
for (int i = 0; i< list.size();i++) {
   Integer val= list.get(i);           // fara conversie de tip !
   sum += val.intValue();
}
```

Exemplu de utilizare dictionar cu valori multiple pentru problema listei de referinte încrucisate - în ce linii dintr-un fisier text apare fiecare cuvânt distinct :

```
public static void main (String[ ] arg) throws IOException {
  Map<String,List<Integer>> cref = new TreeMap<String,List<Integer>>( );
  BufferedReader in = new BufferedReader (new FileReader (arg[0]));
  String line, word;
  int nl=0;
  while ((line=in.readLine()) != null) {
    ++nl;
    StringTokenizer st = new StringTokenizer (line);
    while ( st.hasMoreTokens() ) {
      word=st.nextToken();
      List<Integer> lst = cref.get (word);
      if (lst==null)
        cref.put (word, lst=new LinkedList<Integer>());
      lst.add (new Integer (nl));
    }
```

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

Trecerea de la un tip primitiv la tipul clasã corespunzãtor si invers se face automat în versiunea 1.5, procese numite “autoboxing” si “unboxing”. Exemplu:

```
ArrayList<Integer> list = new ArrayList<Integer>();
for (int i=1;i<10;i++)
    list.add(10*i);              // trecere automata de la int la Integer
 int sum=0;                      // suma valorilor din vector
 for (Iterator i = list.iterator(); i.hasNext();)
      sum += i.next();           // trecere automata de la Integer la int
```

Tot din versiunea 1.5 se poate utiliza o formã mai simplã pentru enumerarea elementelor unei colectii care implementeazã interfata *Iterable*. Exemplu:

```
for (Integer it : list)       // it este iterator pe colectia list de obiecte Integer
     sum += it;          // sau  sum += (int) it;  sau   sum += it.intValue();
```

Un alt exemplu, cu o listã de siruri :

```
   LinkedList<String> list = new LinkedList<String>();
   for (int x=1;x<9;x++)
     list.add(x+"");
   String sb="";
   for (String s : list)
      sb += s;              // concatenare de siruri extrase din vector
```

In versiunea 1.5 toate interfetele si clasele colectie au fost redefinite pentru a permite specificarea tipului elementelor colectiei. Pentru exemplificare redãm un fragment din clasa *ArrayList*, unde “E” este tipul neprecizat al elementelor colectiei :

```
public class ArrayList<E> extends AbstractList<E>
       implements List<E>, RandomAccess, Cloneable, java.io.Serializable {
   private transient E[] elementData;
   private int size;
   public E get(int index) {
     RangeCheck(index);
     return elementData[index];
   }
   public boolean add(E o) {
     ensureCapacity(size + 1);     // Increments modCount!!
     elementData[size++] = o;
     return true;
   }
   public boolean contains(Object elem) {
     return indexOf(elem) >= 0;
   }
   . . .
```

```
}
```

Este posibilã în continuare utilizarea claselor colectie ca înaintea versiunii 1.5, cu elemente de tip *Object*.

A mai fost adãugatã interfata Queue, cu câteva metode noi, interfatã implementatã de mai multe clase, printre care *AbstractQueue, PriorityQueue si LinkedList*.

```
public interface Queue<E> extends Collection<E> {
   boolean offer(E o);  // adauga element la coada (false daca nu mai este loc)
   E poll();          // scoate primul element din coada (null daca nu exista)
   E remove();        // scoate primul element din coada (exceptie daca nu exista)
   E peek();        // primul element din coada (null daca coada goala)
   E element();       // primul element din coada (exceptie daca coada goala)
}
```

Interfetele si clasele de tip coadã (ca *BlockingQueue* si *SynchronousQueue*) au fost introduse, alãturi de alte clase, în pachetul java.util.concurrent pentru programare cu fire de executie concurente (paralele).

Tot ca o noutate a apãrut si o altã colectie ordonatã – coada cu prioritãti. Indiferent de ordinea de adãugare la coadã, elementul din fatã (obtinut prin metoda “peek”) este cel cu prioritatea minimã. Prioritatea este determinatã de cãtre metoda “compareTo” a obiectelor introduse în coadã (constructor fãrã argumente) sau de cãtre metoda “compare” a obiectului comparator transmis ca argument la construirea unei cozi. In exemplul urmãtor se adaugã si se scoate dintr-o coadã de arce ordonatã dupã costuri, în ipoteza cã metoda “compareTo” din clasa “Arc” comparã costuri:

```
PriorityQueue<Arc> pq = new PriorityQueue<Arc>();
…
pq.add (new Arc(v,w,cost));        // adauga arc la coada pq
…
  // afisare coada in ordinea prioritatilor (costului arcelor)
 while ( ! pq.isEmpty())
    System.out.println( pq.remove()); // scoate si sterge din coada
```

In clasa Arrays s-au adãugat metode statice pentru transformarea în sir de caractere a continutului unui vector intrinsec unidimensional sau multidimensional cu elemente de orice tip primitiv sau de tip *Object*. Exemplu de folosire:

```
double [ ] a= {1.1, 2.2, 3.3, 4.4};
System.out.println ( Arrays.toString(a));
Double b[ ][ ]= { {0.0, 0.1, 0.2}, {1.0, 1.1, 1.2}, {2.0, 2.1, 2.2} };
System.out.println(Arrays.deepToString(b));
```

De asemenea, s-au mai adaugat câtiva algoritmi generici pentru colectii:

```
int frequency(Collection<?> c, Object o);   // numara aparitiile obiectului o in colectie
boolean disjoint(Collection<?> c1, Collection<?> c2); // daca colectiile sunt disjuncte
```

```
Comparator<T> reverseOrder(Comparator<T> cmp);   // comp. pentru ordine inversa
```

Din versiunea 1.5 s-a introdus tipul de date definit prin enumerare (si cuvântul cheie *enum*), precum si douã colectii performante pentru elemente de un tip enumerat: *EnumSet* si *EnumMap* (implementate prin vectori de biti).

8. Reutilizarea codului în POO

Reutilizarea codului prin compunere

Unul din avantajele programării orientate pe obiecte este posibilitatea de reutilizare simplã si sigurã a unor clase existente în definirea altor clase, fãrã a modifica clasele initiale. Metodele de reutilizare a codului sunt compozitia si derivarea.

O clasã compusã contine ca membri referinte la obiecte din alte clase. Agregarea unor obiecte de tipuri deja definite într-un nou tip de obiect se impune de la sine acolo unde, în programarea clasicã, se definesc tipuri structurã (înregistrare).

In exemplul urmãtor, clasa "Pers" corespunde unei persoane, pentru care se memoreazã numele si data nasterii. Fiecare obiect de tip "Pers" va contine (pointeri la) un obiect de tip *String* si un obiect de tip *Date*.

```
public class Pers {
  private String nume;                          // nume persoana
  private Date nascut;                          // data nasterii
  Pers (String nume, int zi, int luna, int an) {
    this.nume= nume; this.nascut= new Date(luna,zi,an);
  }
  public String toString () { return nume + "  " + nascut.toString();}
  public Date getBDate () { return nascut;}
}
```

Obiectele de tip “Pers” pot fi memorate într-o colectie ca orice alte obiecte derivate din *Object:*

```
class VecPers  {                          // creare si afisare vector de persoane
  public static void main (String[ ] a) {
    Vector lista = new Vector();
    lista.addElement ( new Pers ("unu",24,11,89));
    lista.addElement ( new Pers ("doi",1,11,99)) ;
    System.out.println (lista);
 }
}
```

Pentru clase ale cãror obiecte se memorează în colectii trebuie redefinite metodele "equals", "hashCode" si, eventual, "compareTo" pentru colectii ordonate (sortabile).

Uneori clasa agregat A contine un singur obiect dintr-o altã clasã B, iar motivul acestei relatii este reutilizarea functionalitãtii clasei B în noua clasã A, adicã folosirea unor metode ale clasei B si pentru obiecte ale clasei A. Interfetele publice ale claselor A si B trebuie sã fie destul de diferite, pentru a justifica compozitia în locul derivării.

In exemplul urmãtor se defineste o clasã pentru stive realizate ca liste înlãntuite de obiecte, cu preluarea functiilor de la obiectul continut (de tip *LinkedList*):

```
public class LinkedStack {
  private LinkedList stack;                  // stiva lista (obiect continut)
  public LinkedStack () {                    // constructor
    stack= new LinkedList();
 }
  public Object push (Object obj) {          // metoda clasei LinkedStack
     stack.addFirst (obj); return obj;       // foloseste metoda clasei LinkedList
  }
  public Object pop () {
     return stack.removeFirst();
  }
  public boolean isEmpty () {                // metode cu acelasi nume si tip
     return stack.isEmpty();
  }
  public String toString () {
     return stack.toString();
  }
}
```

De observat cã tipurile "LinkedStack" si *LinkedList* nu sunt compatibile si nu se pot face atribuiri între ele, desi continutul claselor este atât de asemãnãtor. Varianta definirii clasei "LinkedStack" ca o subclasã a clasei *LinkedList* este preferabilã aici, mai ales cã douã dintre metode pot fi mostenite ca atare ("isEmpty" si "toString").

In exemplul urmãtor se defineste o clasa dictionar cu valori multiple, în care fiecare cheie are asociatã o lista de valori. Clasa preia o mare parte din functionalitatea clasei *HashMap*, prin "delegarea" unor operatii cãtre metode din clasa *HashMap*.

```
public class MultiMap {
   HashMap m = new HashMap();
   public void put (Object key, List val) {
     m.put (key, val);                       // apel HashMap.put
   }
   public List get (Object key) {
     return (List) m.get(key);               // apel HashMap.get
   }
   public String toString () {
     String str="";
     Iterator ik=m.keySet().iterator();      // apel HashMap.keySet
     while (ik.hasNext() ) {
       List lst = get(ik.next());            // lista de valori a unei chei
       str = str+ key + " : " + lst.toString() + "\n";
     }
     return str;
   }
   public Set keySet () {
     return m.keySet();
   }
}
```

Reutilizarea codului prin derivare

Solutia specificã POO de adaptare a unei clase A la alte cerinte este definirea unei clase D, derivatã din clasa A, si care modificã functionalitatea clasei A. Nu se acceptã modificarea codului clasei A de cãtre fiecare utilizator care constatã necesitatea unor modificãri, chiar dacã acest cod sursã este disponibil.

Prin derivare se face o adaptare sau o specializare a unei clase mai generale la anumite cerinte particulare fãrã a opera modificãri în clasa initialã. Extinderea unei clase permite reutilizarea unor metode din superclasã, fie direct, fie dupã "ajustãri" si "adaptãri" cerute de rolul subclasei. Superclasa transmite subclasei o mare parte din functiile sale, nefiind necesarã rescrierea sau apelarea metodelor mostenite.

Vom relua exemplul clasei pentru stive realizate ca liste înlãntuite. Operatiile cu o stivã se numesc traditional “push” si “pop”, dar clasa *LinkedList* foloseste alte nume pentru operatiile respective. De aceea, vom defini douã metode noi:

```
public class LinkedStack extends LinkedList {
  public Object push (Object obj) {
    addFirst (obj);
    return obj;
  }
  public Object pop () {
    return removeFirst();
  }
}
```

De observat cã subclasa “LinkedStack” mosteneste metodele "toString", "isEmpty" si altele din clasa *LinkedList*. De asemenea, existã un constructor implicit care apeleazã constructorul superclasei si care initializeazã lista stivã.

O problemã în acest caz ar putea fi posibilitatea utilizatorilor de a folosi pentru obiecte "LinkedStack" metode mostenite de la superclasã, dar interzise pentru stive: citire si modificare orice element din stivã, cãutare în stivã s.a. Solutia este de a redefini aceste metode în subclasã, cu efectul de aruncare a unor exceptii. Exemplu:

```
    public Object remove (int index)  {
        throw new NoSuchMethodError();
    }
```

Iatã si o altã solutie de definire clasei "MultiMap" (dictionar cu valori multiple), pe baza observatiei cã lista de valori asociatã unei chei (de tip *List*) este tot un obiect compatibil cu tipul *Object* si deci se poate pãstra interfata publicã a clasei *HashMap*:

```
public class MultiMap extends HashMap {
  public Object put (Object key, Object val) {
    List lst = (List) get(key);                  // extrage lista de valori asociata cheii key
    List result=lst;                             // rezultatul va fi lista anterioara
    if (lst==null)                               // daca cheia key nu exista in dictionar
      super.put (key, lst=new ArrayList());      // se introduce o pereche cheie-lista
```

```
    lst.add (val);                    // adauga prima valoare la lista
    return result;                    // lista anterioara de valori
  }
}
```

Toate celelalte metode sunt mostenite ca atare de la superclasa *HashMap* : get, iterator, toString. Dacã dorim sã modificam modul de afisare a unui dictionar atunci putem redefini metoda “toString” din clasa “MultiMap”, ca în exemplul anterior.

Comparatie între compozitie si derivare

Compozitia (un B contine un A) se recomandã atunci când vrem sã folosim (sã reutilizãm) functionalitatea unei clase A în cadrul unei clase B, dar interfetele celor douã clase sunt diferite.

Derivarea (un B este un fel de A) se recomandã atunci când vrem sã reutilizãm o mare parte din (sau toatã) interfata clasei A si pentru clasa B. Aceastã cerintã poate fi motivatã prin crearea de tipuri compatibile A si B: putem sã înlocuim o variabilã sau un argument de tipul A printr-o variabilã (sau argument) de tipul B ("upcasting").

Atât derivarea cât si compozitia permit reutilizarea metodelor unei clase, fie prin mostenire, fie prin delegare (prin apelarea metodelor obiectului continut).

De cele mai multe ori metoda de reutilizare a unor clase se impune de la sine, dar uneori alegerea între compozitie si derivare nu este evidentã si chiar a condus la solutii diferite în biblioteci de clase diferite. Un exemplu este cel al claselor pentru vectori si respectiv pentru stive. Ce este o stivã ? Un caz particular de vector sau un obiect diferit care contine un vector ?

In Java clasa *Stack* este derivatã din clasa *Vector*, desi un obiect stivã nu foloseste metodele clasei *Vector* (cu exceptia metodei "toString"). Mai mult, nici nu se recomandã accesul direct la orice element dintr-o stivã (prin metodele clasei *Vector*).

Exemplul urmãtor defineste o clasã "Stiva" care contine un obiect de tip *Vector*:

```
public class Stiva {
   private Vector items;              // vector folosit ca stiva
   public Stiva() {                   // un constructor
       items = new Vector(10);
   }
   public Object push(Object item) {// pune un obiect pe stiva
     items.addElement(item);
        return item;
   }
   public Object pop() {              // scoate obiect din varful stivei
     int n = items.size();            // n = nr de elemente in stiva
     if ( n  ==0 ) return null;
     Object obj = items.elementAt (n - 1);
     items.removeElementAt (n - 1);
     return obj;
   }
```

```
   public boolean isEmpty() {              // daca stiva goala
     return (items.size() == 0)
   }
   public String toString () {
     return items.toString();
   }
}
```

Clasele “Stiva” si “java.util.Stack” sunt numite si clase “adaptor” pentru cã fac trecerea de la un set de metode publice (cele ale clasei *Vector*) la un alt set de metode publice (“push”, “pop” etc.). Clasa *Stack* este numitã si dublu-adaptor (“Two-way Adapter”), pentru cã permite fie folosirea metodelor superclasei *Vector*, fie folosirea metodelor specifice clasei *Stack* (“push”, “pop” s.a.).

Relatia de compunere dintre obiecte poate fi o relatie dinamicã, modificabilã la executie, spre deosebire de relatia staticã de derivare, stabilitã la scrierea programului si care nu mai poate fi modificatã. O clasã compusã poate contine o variabilã de un tip interfatã sau clasã abstractã, care poate fi înlocuitã la executie (la construirea unui obiect, de exemplu) printr-o referintã la un obiect de un alt tip, compatibil cu tipul variabilei din clasã.

Vom relua exemplul cu stiva adãugând un grad de generalitate prin posibilitatea utilizatorului de a-si alege tipul de listã folosit pentru stivã (vector sau lista înlãntuitã):

```
public class StackList {
  private List stack;                          // adresa vector sau lista inlantuita
  public StackList (List list) {               // constructor
    stack=list;                                // retine adresa obiect stiva
  }
  public Object push (Object obj) {
    stack.add (0,obj);
    return obj;
  }
     . . .          // alte metode
}
```

La construirea unui obiect “StackList” trebuie precizat tipul de listã folosit (vector sau altceva):

```
  StackList st1 = new StackList (new ArrayList());          // vector
  StackList st2 = new StackList (new LinkedList());            // lista inlantuita
```

Un alt exemplu este cel al claselor flux de date cu sumã de control, care contin o variabilã interfatã (*Checksum*), care va primi la instantiere adresa obiectului ce contine metoda de calcul a sumei de control. Legãtura dintre un obiect flux si obiectul de calcul a sumei este stabilitã la executie si asigurã o flexibilitate sporitã. Exemplu:

```
public class CheckedOutputStream extends FilterOutputStream {
   private Checksum cksum;       // adresa obiect cu metoda de calcul suma control
   public CheckedOutputStream(OutputStream out, Checksum cksum) {
    super(out);
    this.cksum = cksum;
   }
   public void write(int b) throws IOException {
    out.write(b);
    cksum.update(b);        // metoda din interfata Checksum
   }
    . . .
}
```

Exemplu de utilizare a clasei anterioare:

```
CRC32 Checker = new CRC32();   // clasa care implem. interfata Checksum
CheckedOutputStream out;
out = new CheckedOutputStream (new FileOutputStream("date"), Checker);
while (in.available() > 0) {
   int c = in.read();  out.write(c);
}
```

Mostenire multiplã prin derivare si compozitie

In Java o subclasã nu poate avea decât o singurã superclasã, dar uneori este necesar ca o clasã sã preia functii de la douã sau mai multe clase diferite. Implementarea mai multor interfete de cãtre o clasã nu este o solutie pentru mostenirea multiplã de functii.

O clasã M poate prelua metode de la douã clase A si B astfel: clasa M extinde pe A si contine o variabilã de tip B; metodele din M fie apeleazã metode din clasa B, fie sunt mostenite de la clasa A. Clasa A este de multe ori o clasã abstractã, iar clasa B este instantiabilã sau abstractã. Exemplu de mostenire functii de la 3 clase:

```
class A {
  void f1 () { System.out.println ("A.f1"); }
}
class B {
  void f2 () { System.out.println ("B.f2"); }
}
class C {
  void f3 () { System.out.println ("C.f3"); }
}
class M extends C {
   A a = new A ();       // initializarea var. a si b se poate face intr-un constructor
   B b = new B ();
   void f1 () { a.f1();}    // delegare obiect a pentru operatia f1
   void f2 () { b.f2();}    // delegare obiect b pentru operatia f2
}
class X {
```

```
  public static void main (String arg[]) {
    M m = new M();
    m.f1(); m.f2(); m.f3();
  }
}
```

Un exemplu real de mostenire multiplã poate fi o clasã pentru o multime realizatã ca vector, care extinde clasa *AbstractSet* si contine o variabilã de tip *ArrayList* :

```
public class ArraySet extends AbstractSet {
  private ArrayList set;
  public ArraySet() {
    set = new ArrayList();
  }
  public boolean add (Object obj) {
    if (! set.contains(obj) )
      return set.add(obj);          // delegare pentru operatia de adaugare
    return false;
  }
  public Iterator iterator() {
    return set.iterator();          // delegare pentru creare obiect iterator
  }
  public int size() {
    return set.size();
  }
}
```

Pentru multimi de tipul "ArraySet" se pot folosi toate metodele mostenite de la clasa *AbstractSet*: toString, contains, containsAll, addAll, removeAll, retainAll s.a.

Aceeasi solutie de mostenire multiplã este folositã în câteva clase JFC (Swing) de tip “model”; de exemplu, clasa *DefaultListModel* preia metode de la superclasa *AbstractListModel* si deleagã unei variabile interne de tip *Vector* operatii cu un vector de obiecte. Un model de listã este un vector cu posibilitãti de generare evenimente (de apelare receptori) la modificãri operate în vector.

Combinarea compozitiei cu derivarea

Derivarea pãstreazã interfata clasei de bazã, iar agregarea permite mai multã flexibilitate la executie. Combinarea agregãrii cu derivarea îmbinã avantajele ambelor metode de reutilizare: mostenirea interfetei si posibilitatea modificãrii obiectului interior din obiectul unei clase agregat. Uneori variabila din subclasã este chiar de tipul superclasei. Clasele ce contin un obiect de acelasi tip sau de un tip compatibil cu al superclasei sale se mai numesc si clase “anvelopã” (“wrapper”), deoarece adaugã functii obiectului continut, ca un ambalaj pentru acel obiect. O clasa anvelopã este numitã si clasã “decorator”, deoarece "decoreazã" cu noi functii o clasã existentã.

De exemplu, putem defini o clasã stivã mai generalã, care sã poatã folosi fie un vector, fie o listã înlãntuitã, dupã cum doreste programatorul. Clasa anvelopã care urmeazã este compatibilã cu tipul *List* si, în acelasi timp, foloseste metode definite în clasa *AbstractList* :

```
class StackList extends AbstractList {
  private AbstractList stack;                  // adresa obiect stiva
  public StackList (List list) {               // constructor
    stack=(AbstractList)list;                  // retine adresa obiect stiva
  }
  public Object push (Object obj) {
    stack.add (0,obj);
    return obj;
  }
  public Object pop () {
    Object obj= get(0);
    stack.remove(obj);
    return obj;
  }
  public int size() { return stack.size(); }
}
```

Exemple de combinare a derivãrii si compozitiei se gãsesc în clasa *Collections*, pentru definirea de clase colectie cu functionalitate putin modificatã fatã de colectiile uzuale, dar compatibile cu acestea ca tip.

Primul grup de clase este cel al colectiilor nemodificabile, care diferã de colectiile generale prin interzicerea operatiilor ce pot modifica continutul colectiei. Exemplu:

```
class UnmodifiableCollection implements Collection, Serializable {
    Collection c;
        // metode care nu modifica continutul colectiei
    public int size()    {return c.size();}
    public boolean isEmpty()      {return c.isEmpty();}
    public boolean contains(Object o)   {return c.contains(o);}
    public String toString()           {return c.toString();}
    . . .
      // metode care ar putea modifica continutul colectiei
    public boolean add (Object o){
       throw new UnsupportedOperationException();
    }
    public boolean remove (Object o) {
       throw new UnsupportedOperationException();
    }
    . . .
}
```

Incercarea de adãugare a unui nou obiect la o colectie nemodificabilã produce o exceptie la executie; dacã nu se defineau metodele "add", "remove" s.a. în clasa

derivatã, atunci apelarea metodei "add" era semnalatã ca eroare la compilare. Exceptia poate fi tratatã fãrã a împiedica executia programului.

Al doilea grup de clase sunt clasele pentru colectii sincronizate. Exemplu:

```
class SynchronizedCollection implements Collection, Serializable {
    Collection c;                    // colectia de baza
    public int size() {
       synchronized(this) {return c.size();}
   }
    public boolean add(Object o) {
       synchronized(this) {return c.add(o);}
    }
    . . .  // alte metode
}
```

In realitate, clasele prezentate sunt clase incluse statice si sunt instantiate în metode statice din clasa *Collections*:

```
    static class UnmodifiableList extends UnmodifiableCollection implements List {
        private List list;
        UnmodifiableList(List list) {
           super(list);   this.list = list;
        }
        . . .   // alte metode
    }
    public static List unmodifiableList (List list) {
        return new UnmodifiableList(list);
    }
```

Exemple de utilizare a claselor colectie "speciale" :

```
    String kw[] ={"if", "else", "do","while", "for"};
    List list = Collections.unmodifiableList (Arrays.asList(kw));
    Set s = Collections.synchronizedSet (new HashSet());
```

Clase decorator de intrare-iesire

Combinarea derivãrii cu delegarea a fost folositã la proiectarea claselor din pachetul “java.io”. Existã douã familii de clase paralele : familia claselor “flux” (“Stream”) cu citire-scriere de octeti si familia claselor “Reader-Writer”, cu citire-scriere de caractere.

Numãrul de clase instantiabile de I/E este relativ mare deoarece sunt posibile diverse combinatii între suportul fizic al fluxului de date si facilitãtile oferite de fluxul respectiv. Toate clasele flux de intrare sunt subtipuri ale tipului *InputStream* (*Reader*) si toate clasele flux de iesire sunt subtipuri ale tipului *OutputStream* (*Writer*). Clasele *Reader, Writer* si celelalte sunt clase abstracte.

Dupã suportul fizic al fluxului de date se poate alege între:
- Fisiere disc : *FileInputStream, FileOutputStream, FileReader, FileWriter* s.a.
- Vector de octeti sau de caractere : *ByteArrayInputStream, ByteArrayOutputStream CharArrayReader, CharArrayWriter*.
- Buffer de siruri (în memorie): *StringBufferInputStream, StringBufferOutputStream*.
- Canal pentru comunicarea sincronizatã între fire de executie : *PipedInputStream, PipedOutputStream, PipedReader, PipedWriter*.

Dupã facilitãtile oferite avem de ales între:
- Citire-scriere la nivel de octet sau bloc de octeti (metode "read", "write").
- Citire-scriere pe octeti dar cu zonã buffer: *BufferedInputStream, BufferedReader, BufferedOutputStream, BufferedWriter*.
- Citire-scriere la nivel de linie si pentru numere de diferite tipuri (fãrã conversie): *DataInputStream, DataOutputStream*..
- Citire cu punere înapoi în flux a ultimului octet citit (*PushBackInputStream*).
- Citire însotitã de numerotare automatã a liniilor citite (*LineNumberInputStream*).

Combinarea celor 8 clase sursã/destinatie cu optiunile de prelucrare asociate transferului de date se face prin intermediul claselor anvelopã , care sunt numite si clase "filtru" de intrare-iesire. Dacã s-ar fi utilizat derivarea pentru obtinerea claselor direct utilizabile atunci ar fi trebuit generate, prin derivare, combinatii ale celor 8 clase cu cele 4 optiuni, deci 32 de clase (practic, mai putine, deoarece unele optiuni nu au sens pentru orice flux). Pentru a folosi mai multe optiuni cu acelasi flux ar fi trebuit mai multe niveluri de derivare si deci ar fi rezultat un numãr si mai mare de clase.

Solutia claselor anvelopã permite sã se adauge unor clase de bazã diverse functii în diferite combinatii. Principalele clase filtru de intrare-iesire sunt:

FilterInputStream, FilterOutputStream si *FilterReader, FilterWriter*.

Clasele de tip filtru sunt clase intermediare, din care sunt derivate clase care adaugã operatii specifice (de "prelucrare"): citire de linii de text de lungime variabilã, citire-scriere de numere în format intern, scriere numere cu conversie de format s.a. O clasã anvelopã de I/E contine o variabilã de tipul abstract *OutputStream* sau *InputStream*, care va fi înlocuitã cu o variabilã de un tip flux concret (*FileOutputStream*, ...), la construirea unui obiect de un tip flux direct utilizabil.

Clasa decorator *FilterInputStream* este derivatã din *InputStream* si, în acelasi timp, contine o variabilã de tip *InputStream*:

```
public class FilterInputStream extends InputStream {
  protected InputStream in;
  protected FilterInputStream (InputStream in) {        // constructor (in clasa abstracta)
     this.in=in;                 // adresa obiectului "flux"
  }
    // citirea unui octet
  public int read () throws IOException {
     return in.read ();    // citirea depinde de tipul fluxului
  }
  ...  // alte metode
}
```

Metoda "read" este o metodã polimorficã, iar selectarea metodei necesare se face în functie de tipul concret al variabilei "in" (transmis ca argument constructorului). Nu se pot crea obiecte de tipul *FilterInputStream* deoarece constructorul clasei este de tip *protected*. In schimb, se pot crea obiecte din subclase ale clasei *FilterInputStream*.

Clasele *DataInputStream*, *BufferedInputStream*, *LineNumberInputStream* si *PushbackInputStream* sunt derivate din clasa *FilterInputStream* si sunt clasele de prelucrare a datelor citite. Cea mai folositã este clasa *DataInputStream* care adaugã metodelor de citire de octeti mostenite si metode de citire a tuturor tipurilor primitive de date: "readInt", "readBoolean", readFloat", "readLine", etc.

La crearea unui obiect de tipul *DataInputStream* constructorul primeste un argument de tipul *InputStream*, sau un tip derivat direct din *InputStream*, sau de un tip derivat din *FilterInputStream*. Pentru a citi linii dintr-un fisier folosind o zonã tampon, cu numerotare de linii vom folosi urmãtoarea secventã de instructiuni:

```
public static void main (String arg[]) throws IOException {
   FileInputStream fis= new FileInputStream (arg[0]);
   BufferedInputStream bis = new  BufferedInputStream (fis);
   LineNumberInputStream lnis= new LineNumberInputStream (bis);
   DataInputStream dis = new DataInputStream (lnis);
   String linie;
   while ( (linie=dis.readLine()) != null)
     System.out.println (lnis.getLineNumber()+" "+linie);
}
```

De obicei nu se mai folosesc variabile intermediare la construirea unui obiect flux. Exemplu de citire linii , cu buffer, dintr-un fisier disc:

```
public static void main (String arg[ ]) throws IOException {
  DataInputStream dis = new DataInputStream (
                              new BufferedInputStream (new FileInputStream (arg[0])));
  String linie;
  while ( (linie=dis.readLine()) != null)
     System.out.println ( linie);
}
```

Ordinea în care sunt create obiectele de tip *InputStream* este importantã : ultimul obiect trebuie sã fie de tipul *DataInputStream*, pentru a putea folosi metode ca "readLine" si altele. Si familia claselor *Reader-Writer* foloseste clase decorator:

```
public abstract class FilterReader extends Reader {
   protected Reader in;
   protected FilterReader(Reader in) {  super(in); this.in = in;  }
   public int read() throws IOException { return in.read();    }
   public int read(char cbuf[ ], int off, int len) throws IOException {
            return in.read(cbuf, off, len);
  }
   public void close() throws IOException {
```

```
    in.close();
   }
}
```

Clasele *PrintStream* si *PrintWriter* adaugã claselor filtru metode pentru scriere cu format (cu conversie) într-un flux de iesire, metode cu numele “print” sau “println”. Constanta “System.out” este de tipul *PrintStream* si corespunde ecranului, iar constanta “System.in” este de un subtip al tipului *InputStream* si corespunde tastaturii.

Clasa *BufferedReader* adaugã clasei *Reader* o metodã “readLine” pentru a permite citirea de linii din fisiere text.

Clasele *InputStreamReader* si *OutputStreamWriter* sunt clase adaptor între clasele “Stream” si clasele “Reader-Writer”. Clasele adaptor extind o clasã *Reader* (*Writer*) si contin o variabilã de tip *InputStream* (*OutputStream*); o parte din operatiile impuse de superclasã sunt realizate prin apelarea operatiilor pentru variabila flux (prin “delegare”). Este deci un alt caz de combinare între extindere si agregare.

Un obiect *InputStreamReader* poate fi folosit la fel ca un obiect *Reader* pentru citirea de caractere, dar în interior se citesc octeti si se convertesc octeti la caractere, folosind metode de conversie ale unei clase convertor. Codul urmãtor ilustreazã esenta clasei adaptor, dar sursa clasei prevede posibilitatea ca un caracter sã fie format din doi sau mai multi octeti (conversia se face pe un bloc de octeti):

```
public class InputStreamReader extends Reader {
   private ByteToCharConverter btc;
   private InputStream in;
   private InputStreamReader(InputStream in, ByteToCharConverter btc) {
     super(in);
     this.in = in;  this.btc = btc;
   }
   public InputStreamReader(InputStream in) {
      this(in, ByteToCharConverter.getDefault());
   }
   public int read() throws IOException {
       int byte = in.read();                  // citire octet
       char ch = btc.convert(byte);           // conversie din byte in char
       return ch;                             // caracter coresp. octetului citit
   }
  . . .     // alte metode
}
```

Exemplu de utilizare a unei clase adaptor pentru citire de linii de la tastaturã :

```
public static void main (String arg[ ]) throws IOException {
  BufferedReader stdin = new BufferedReader(new InputStreamReader (System.in));
  String line;
  while ( (line=stdin.readLine()) != null) {          // dacã nu s-a introdus ^Z (EOF)
     System.out.println (line);
  }
```

```
}
```

9. Clase incluse

Clase incluse

O clasã Java poate contine, ca membri ai clasei, alte clase numite clase incluse (“nested classes”). In cazul unei singure clase incluse, structura va arãta astfel:

```
public class Outer {
   . . .                    // date si/sau metode ale clasei Outer
     public class Inner {
         . . .              // date si/sau metode ale clasei Inner
     }
   . . .                    // alti membri ai clasei Outer
}
```

Un bun exemplu este o clasã iterator inclusã în clasa colectie pe care actioneazã. In acest fel clasa iterator are acces la variabile *private* ale colectiei, nu poate fi instantiatã direct ci numai prin intermediul clasei (nu poate exista obiect iterator fãrã o colectie), fiind ascunsã altor clase. Exemplu de iterator pe vector, clasã interiarã:

```
public class Vector extends AbstractList implements List, Cloneable {
  protected int count;                    // nr de elemente in vector
  protected Object elementData[];         // vector de obiecte
  // metoda care produce obiect iterator
  public Enumeration elements() {
     return new VectorEnumeration()
  }
     // definirea clasei iterator pe vector (inclusa)
  class VectorEnumeration implements Enumeration  {
    int count = 0;          // indice element curent din enumerare
    public boolean hasMoreElements() {
        return count < elementCount;
    }
    public Object nextElement() {
         if (count < elementCount)
             return elementData[count++];
    }
  }
      // . . .   alte metode din clasa Vector
}
```

Ca orice alt membru, clasa inclusã poate fi declaratã *public* sau *private* si poate fi staticã (cu existentã independentã de instantele clasei exterioare).

O clasã inclusã nestaticã este numitã si clasã interioarã (“inner class”), pentru cã fiecare obiect din clasa exterioarã va include un obiect din clasa interioarã.

O altã formã de clasã interioarã este o clasã definitã într-o metodã a clasei externe. Exemplu de clasã pentru un obiect comparator inclusã în functia de ordomare:

```
  // ordonarea unui dictionar în ordinea valorilor asociate cheilor
static void sortByValue (Map m) {
        // clasa inclusa
  class Comp implements Comparator {            // compara doua perechi cheie-val
   public int compare (Object o1, Object o2) {
     Map.Entry  e1= (Map.Entry)o1;                    // o pereche cheie-valoare
     Map.Entry  e2= (Map.Entry)o2;                    // alta pereche cheie-valoare
     return ((Integer)e2.getValue()).intValue() -     // compara valori
          ((Integer) e1.getValue()).intValue();
   }
 }
   Set eset = m.entrySet();                     // multime de perechi cheie-valoare
   ArrayList entries = new ArrayList(eset);     // vector de perechi cheie-valoare
   Collections.sort (entries, new Comp());      // ordonare vector
   System.out.println (entries);                // afisare perechi ordonate dupa valori
}
```

Toate aceste clase sunt definite explicit, folosind cuvântul *class*.

In plus, se pot defini ad-hoc clase incluse anonime, pe baza unei alte clase sau a unei interfete (prin extindere sau prin implementare implicitã, deoarece nu se folosesc cuvintele *extends* sau *implements*).

Motivele definirii de clase interioare pot fi diverse:

- Pentru clase de interes local : clasa interioarã este necesarã numai clasei exterioare.
- Pentru reducerea numãrului de clase de nivel superior si deci a conflictelor între nume de clase (ca urmare a unor instructiuni “import” pentru pachete în care se aflã clase cu nume identice).
- Pentru a permite clasei exterioare accesul la membri *private* ai clasei interioare.
- Pentru a permite claselor interioare accesul la variabile ale clasei exterioare si deci o comunicare mai simplã între clasele incluse (prin variabile externe lor).
- Pentru ca o clasã sã poatã mosteni functii de la câteva clase (mostenire multiplã).

Clase incluse cu nume

Clasele incluse cu nume primesc de la compilator un nume compus din numele clasei exterioare, caracterul ‘$’ si numele clasei interioare. Clasele care nu sunt incluse în alte clase se numesc clase de nivel superior (“top-level classes”).

In exemplul urmãtor o clasã pentru un vector ordonat ("SortedArray") contine o variabilã comparator ("cmp"), care poate fi initializatã de un constructor al clasei. Dacã se foloseste constructorul fãrã argumente, atunci variabila comparator primeste o valoare implicitã, ca referintã la un obiect comparator dintr-o clasã interioarã. Clasa "DefaultComp" nu mai trebuie definitã de utilizatori si transmisã din afarã, ea este utilã numai clasei în care este definitã (clasa exterioarã "SortedArray"):

```
public class SortedArray extends ArrayList {
     // clasa interioara
  private class DefaultComp implements Comparator {
```

```
    public int compare (Object e1, Object e2) {
      Comparable c1=(Comparable)e1;
      Comparable c2=(Comparable)e2;
      return c1.compareTo(c2);
    }
  }
      // alti membri ai clasei SortedArray
  Comparator cmp=new DefaultComp();                    // comparator implicit
  public SortedArray () { super();}
  public SortedArray (Comparator comp) {
    super();
    cmp=comp;
  }
  public boolean add (Object obj) {
    int k=indexOf(obj);
    if (k < 0) k= -k-1;
    super.add(k, obj);
    return true;
  }
  public int indexOf (Object obj) {
    return Collections.binarySearch (this,obj,cmp);
  }
```

In exemplul urmãtor clasa inclusã “Entry” este folositã numai în definitia clasei “ArrayMap”. Clasa inclusã “Entry”implementeazã o interfatã inclusã (interfata *Entry* inclusã în interfata *Map* este publicã si deci utilizabilã din orice altã clasã):

```
public class ArrayMap extends AbstractMap {
      // clasa interioara
  static class Entry implements Map.Entry {
    private Object key,val;
    public Entry (Object k, Object v) {
      key=k; val=v;
    }
    public String toString() { return key+"="+val;}
    public Object getKey() { return key; }
    public Object getValue() { return val;}
    public Object setValue (Object v) { val=v; return v;}
  }
       // alti membri ai clasei ArrayMap
  private Vector entries ;                          // perechi cheie-valoare
  public ArrayMap (int n) {                         // constructor
     entries = new Vector(n);
  }
  public Object put ( Object key, Object value) {
    entries.addElement (new Entry(key,value));
    return key;
  }
   public Set entrySet () {
    HashSet set = new HashSet();
```

```
    for (int i=0;i<entries.size();i++)
      set.add(entries.get(i));
    return set;
  }
}
```

Tipul *Entry* este folosit în mai multe metode ale clasei *AbstractMap*. Exemplu:

```
public Object get(Object key) {
   Iterator i = entrySet().iterator();
   while (i.hasNext()) {
     Entry e = (Entry) i.next();
     if (key.equals(e.getKey()))
         return e.getValue();
   }
   return null;
}
```

Clasa interioarã staticã *ReverseComparator* din clasa *Collections*, este folositã de metoda staticã “reverseOrder” prin intermediul unei variabilei statice:

```
    public static Comparator reverseOrder() {        // din clasa Collections
       return REVERSE_ORDER;
    }
    private static final Comparator REVERSE_ORDER = new ReverseComparator();
   private static class ReverseComparator implements Comparator,Serializable {
     public int compare(Object o1, Object o2) {
           Comparable c1 = (Comparable)o1;
           Comparable c2 = (Comparable)o2;
           return -c1.compareTo(c2);
       }
    }
```

Simplificarea comunicãrii între clase

In exemplul urmãtor metodele clasei exterioare “SimpleList” se pot referi direct la variabile din clasa inclusã “Node” (si nu prin intermediul unor metode publice).

```
public class SimpleList extends AbstractList {      // clasa pentru liste simplu inlantuite
  // clasa inclusa in clasa SimpleList
class Node {
  private Object val;
  private Node next;
  public Node (Object ob) {  val=ob; next=null; }
}
    // variabile ale clasei SimpleList
  private Node head;                    // inceput lista
  private int n;                        // nr de noduri in lista
    // functii membre ale clasei SimpleList
```

```
  public SimpleList () {
    head= new Node(null);              // santinela
    n=0;
  }
  public int size() {  return n; }
  public Object get (int k) {
    if ( k > n) return null;
    Node p=head.next;                  // var. din clasa inclusa
    for (int i=0;i<k;i++)
      p=p.next;                        // var. din clasa inclusa
    return p.val;                      // var. din clasa inclusa
  }
public boolean add (Object el) {
     // adauga la sfarsit daca nu exista deja
     Node nou = new Node(el);
     Node p =head;                     // var. din clasa inclusa
     while (p.next != null)            // var. din clasa inclusa
       p=p.next;                       // var. din clasa inclusa
     p.next=nou;                       // var. din clasa inclusa
     n=n+1;
     return true;
  }
}
```

O clasã iterator poate lucra cu variabile ale clasei colectie pe care face enumerarea; accesul direct la aceste variabile este simplificat dacã se include clasa iterator în clasa colectie. Instantierea clasei interioare se face prin metoda “iterator” din clasa exterioarã:

```
public class SimpleList extends AbstractList {
  private Node head;                   // inceput lista
     // clasa iterator inclusa
  class SListIterator implements Iterator {
        private Node pos;              // cursor= adresa nod curent
     SListIterator () {                // constructor
        pos=head.next;                 // var “head” din clasa SimpleList
     }
     public boolean hasNext () {
       return pos != null;
     }
     public Object next() {
       Object obj =pos.val;            // var. din clasa Node
       pos=pos.next;                   // var. din clasa Node
       return obj;
     }
     public void remove () {
       throw new UnsupportedOperationException();
    }
  }
```

```
        // metoda a clasei SimpleList
    public Iterator iterator () {
        return new SListIterator();
 }
  ...        // alti membri ai clasei exterioare: clase, variabile, metode  }
```

Clasele interioare cu date comune

O clasã inclusã într-o altã clasã este un membru al acelei clase si are acces la ceilalti membri ai clasei (variabile si metode), variabilele fiind componente ale instantei curente.

Includerea a douã clase A si B într-o aceeasi clasã C permite simplificarea transmiterii de date între clasele A si B prin variabile ale clasei C. O astfel de situatie apare în cazul a douã clase cu rol de producãtor si respectiv de consumator care-si transmit date printr-o coadã de obiecte, pentru acoperirea diferentei dintre ritmul de producere si ritmul de consumare a mesajelor. Obiectul coadã este folosit atât de obiectul producãtor cât si de obiectul consumator. Avem cel putin douã solutii pentru a permite accesul la obiectul comun coadã:

- Transmiterea unei referinte la acest obiect la construirea obiectelor producãtor si consumator .
- Includerea claselor producãtor si consumator într-o clasã gazdã, alãturi de obiectul coadã.

Urmeazã mai întâi solutia cu clase de acelasi nivel:

```
class Producer {                                    // proces producator
   Queue q;
   public Producer (Queue q) { this.q=q;  }
   public void put(Object x) {                      // pune un obiect in coada
     q.add(x);
   }
 }
 class Consumer {                                   // proces consumator
   Queue q;
   public Consumer (Queue q) { this.q=q; }
   public Object get() {
     if ( q.isEmpty())  return null;                // daca coada goala
      return q.del();
   }
 }
    // simulare procese producator-consumator
class Simulare {
   public static void main(String[] args) throws Exception {
     int qm=0,ql, r, k=0; Integer x;
     Queue q= new Queue();
     Producer p=new Producer(q);            // obiectul p se refera la ob. q
     Consumer c=new Consumer(q);            // obiectul c se refera la ob. q
     while ( k < 21) {
       r= ((int)(Math.random() * 2));
```

```
      switch (r) {
        case 0:  p.put(new Integer(k)); k++; break;      // activare producator
        case 1:  System.out.println(c.get()); break;     // activare consumator
      }
}
```

Pentru comparatie urmeazã solutia cu clase interioare. Clasele incluse au fost declarate statice pentru a putea fi instantiate din metoda staticã “main”.

```
class Simulare {
  static class Producer {             // clasa inclusa
    public void run() {
      q.add(new Byte(k)); k++;
    }
  }
  static class Consumer {             // clasa inclusa
    public void run() {
       if ( ! q.isEmpty())
         System.out.println("Consumator " + " scoate " + q.del());
    }
  }
  static Queue q ;                        // obiectul coada
  static Producer p ;                     // proces producator
  static Consumer c ;                     // proces consumator
  static byte k=1;
     // simulare procese producator-consumator
    public static void main(String[] args) {
      q= new Queue();
      p=new Producer(); c=new Consumer();
      while ( k <=20)
        switch ((int)(Math.random()*2)) {
          case 0:  p.run(); break;
          case 1:  c.run(); break;
        }
    }
}
```

Clase interioare anonime

Uneori numele unei clase incluse apare o singurã datã, pentru a crea un singur obiect de acest tip. In plus, clasa inclusã implementeazã o interfatã sau extinde o altã clasã si contine numai câteva metode scurte. Pentru astfel de situatii se admite definirea ad-hoc de clase anonime, printr-un bloc care urmeazã operatorului *new* cu un nume de interfatã sau de clasã abstractã.

Sintaxa definirii unei clase anonime este urmãtoarea:

```
    new Interf ( ) { ... // definitie clasa inclusa } ;
```

unde "Interf" este un nume de interfatã (sau de clasã abstractã sau neabstractã) din care este derivatã (implicit) clasa inclusã anonimã.

O astfel de clasã nu poate avea un constructor explicit si deci nu poate primi date la construirea unui obiect din clasa anonimã.

O situatie tipicã pentru folosirea unei clase anonime definitã simultan cu crearea unui obiect de tipul respectiv este cea în care transmitem unei functii un obiect de un subtip al interfetei *Comparator* (adresa unei functii de comparare). Exemplu de sortare a unei liste de obiecte în ordine descrescãtoare :

```
Collections.sort (list, new Comparator( ) {            // ordonare descrescatoare
   public int compare (Object t1, Object t2) {         // incepe definitia clasei anonime
      Comparable c1=(Comparable)t1, c2=(Comparable)t2;
      return - c1.compareTo(c2);                       // rezultat invers metodei compareTo
   }
 });                                          // aici se termina definitia clasei si instructiunea
```

Alt exemplu de clasã comparator definitã ca o clasã interioarã anonimã:

```
class SortedArray extends ArrayList {
  private Comparator cmp=new Comparator () {        // comparator implicit
    public int compare (Object e1, Object e2) {
      Comparable c1=(Comparable)e1;
      Comparable c2=(Comparable)e2;
      return c1.compareTo(c2);
    }
  };
  public SortedArray () { super();}
  public SortedArray (Comparator comp) { super(); cmp=comp; }
  public int indexOf (Object obj) {
    return Collections.binarySearch (this,obj,cmp);
  }
    ...  // alte metode
}
```

Iatã si o altã definitie a metodei “iterator” din clasa “SimpleList”, în care clasa iterator pentru liste este anonimã si este definitã în expresia care urmeazã lui *new*.

```
public Iterator iterator () {
    return new Iterator() {
        // definirea clasei anonime iterator  ca o clasa inclusa
      private Node pos=head.next;
      public boolean hasNext () {
        return pos != null;
      }
      public Object next() {
        Object obj =pos.val;
        pos=pos.next;
        return obj;
      }
```

```
    public void remove () {
    }
  };  // sfârsit instructiune return new Iterator ( )
}     // sfarsit metoda iterator
```

Prin definirea de clase anonime codul sursã devine mai compact iar definitia clasei apare chiar acolo unde este folositã, dar pot apare dificultãti la întelegerea codului si erori de utilizare a acoladelor si parantezelor.

Intre clasa interioarã si clasa exterioarã existã un "cuplaj" foarte strâns; acest cuplaj poate fi un dezavantaj la restructurarea (refactorizarea) unei aplicatii, dar poate fi exploatat în definirea unor clase de bibliotecã (care nu se mai modificã). Exemplu:

```
public abstract class AbstractMap implements Map {
  public Collection values() {            // o colectie a valorilor din dictionar
    if (values == null) {                 // ptr apeluri repetate ale metodei values
       values = new AbstractCollection() {  // subclasa interioara anonima
        public Iterator iterator() {
           return new Iterator() {        // alta clasa interioara anonima
             private Iterator i = entrySet().iterator();
             public boolean hasNext() {
               return i.hasNext();
             }
             public Object next() {
               return ((Entry)i.next()).getValue();
             }
             public void remove() {
               i.remove();
             }
           };         // aici se termina instr. return new Iterator …
       }
        public int size() {          // din subclasa lui AbstractCollection
           return AbstractMap.this.size();  // this = obiect din clasa anonima
        }
        public boolean contains(Object v) {
           return AbstractMap.this.containsValue(v);
        }
     };   // aici se termina instr. values= new …
    }
    return values;
   }
}
```

Mostenire multiplã prin clase incluse

O clasã interioarã poate mosteni de la o implementare (nu de la o interfatã) în mod independent de mostenirea clasei exterioare si de mostenirile altor clase incluse. Aceste mosteniri se pot combina în obiecte ale clasei exterioare, pentru care se pot folosi metode mostenite de toate clasele incluse. Exemplu:

```
class A {  void f1 () { System.out.println ("A.f1"); }  }
```

```
class B {  void f2 () { System.out.println ("B.f2"); }  }
class C {  void f3 () { System.out.println ("C.f3"); }  }
class M extends C {
   class AltA extends A { }
   class AltB extends B { }
   void f1 () { new AltA().f1(); }
   void f2 () { new AltB().f2(); }
}
class X {
  public static void main (String arg[]) {
    M m = new M();
    m.f1(); m.f2(); m.f3();
  }
}
```

Pentru clasa M se pot apela functii mostenite de la clasele A, B si C la care se pot adãuga si alte functii suplimentare.

Clasele "AltA" si "AltB" nu sunt folosite decât în functiile "f1" si "f2", deci am putea folosi clase incluse anonime astfel:

```
class M extends C {
   A makeA() { return new A() {}; }
   void f1 () { makeA().f1(); }
   B makeB() { return new B() {}; }
   void f2 () { makeB().f2(); }
}
```

Acelasi efect se poate obtine si prin combinarea derivării cu compozitia:

```
class M extends C {
    A a = new A ();   B b = new B ();
    void f1 () { a.f1();}
    void f2 () { b.f2();}
}
```

Solutia claselor incluse oferã în plus posibilitatea folosirii unor obiecte de tip M la apelarea unor functii cu argumente de tip A sau B (si C, superclasa lui M):

```
class X {
  static void useA (A a) { a.f1(); }          // functie cu argument de tip A
  static void useB (B b) { b.f2(); }          // functie cu argument d tip B
  static void useC (C c) { c.f3(); }          // functie cu argument d tip B
  public static void main (String arg[ ]) {
    M m = new M();
    m.f1(); m.f2(); m.f3();
    useA (m.makeA());
    useB (m.makeB());
    useC (m);
```

```
  }
}
```

Probleme asociate claselor incluse

O clasã inclusã într-un bloc poate folosi variabile locale din acel bloc, inclusiv argumente formale ale functiei definite prin blocul respectiv. Orice variabilã sau parametru formal folosit într-o clasã inclusã trebuie declarat cu atributul *final*, deoarece aceastã variabilã este copiatã în fiecare instantã a clasei incluse si toate aceste copii trebuie sã aibã aceeasi valoare (nemodificatã în cursul executiei). Exemplul urmãtor este o functie similarã functiei “iterator” dintr-o clasã colectie, dar itereazã pe un vector intrinsec de obiecte. In prima variantã a acestei functii se defineste o clasã cu nume interioarã unui bloc si care foloseste un argument al functiei care contine definitia clasei.

```
    // functie care produce un iterator pentru vectori intrinseci
 public static Iterator arrayIterator (final Object a[] ) {
    // clasa interioara functiei
   class AI implements Iterator {
     int i=0;
    public boolean hasNext() {
        return i < a.length;
    }
    public Object next() {
       return a[i++];
    }
    public void remove() { }    // neimplementata
   }
   return new AI();
 }
```

Clasa AI poate sã fie definitã ca o clasã inclusã anonimã deoarece acest nume nu este folosit în afara functiei “arrayIterator”. O clasã definitã într-un bloc nu este membrã a clasei ce contine acel bloc si nu este vizibilã din afara blocului. Exemplu de clasã anonimã definitã într-un bloc:

```
    // iterator pentru vectori intrinseci
   public static Iterator arrayIterator (final Object a[] ) {
     return new Iterator () {          // urmeaza definitia clasei anonime
         int i=0;
         public boolean hasNext() {
             return i < a.length;
         }
         public Object next() {
             return a[i++];
         }
         public void remove() { throw new UnsupportedOperationException();}
```

```
    };   // aici se termina instr. return
  }
```

O clasã anonimã nu poate avea un constructor explicit si poate fie sã extindã o altã clasã, fie sã implementeze o interfatã (ca în exemplul anterior) cu aceeasi sintaxã. Un nume de interfatã poate urma cuvântului cheie *new* numai la definirea unei clase anonime care implementeazã acea interfatã, iar lista de argumente trebuie sã fie vidã.

O clasã anonimã nu poate simultan sã extindã o altã clasã si sã implementeze o interfatã, sau sã implementeze mai multe interfete.

Numele claselor incluse anonime sunt generate de compilator prin adãugarea la numele clasei exterioare a caracterului '$'si a unui numãr (a câta clasã inclusã este).

Variabilele locale din clasa exterioarã sau din blocul exterior sunt copiate de compilator în câmpuri *private* ale clasei interioare, cu nume generate automat de compilator si transmise ca argumente constructorului clasei interioare. Pentru functia anterioarã compilatorul Java genereazã un cod echivalent de forma urmãtoare:

```
public static Iterator arrayIterator (final Object a[]) {
    return new C$1(a);          // C este numele clasei ce contine metoda
}
class C$1 implements Iterator {
    private Object val$a[];
    int i;
    OuterClass$1 (Object val$a[]) {      // constructor
        this.val$a = val$a; i=0;
    }
    public boolean hasNext () {
        return i < val$a.length;
    }
    public Object next() {
        return val$a[i++];
    }
    public void remove() { throw new UnsupportedOperationException();}
}
```

10. Clase pentru o interfatã graficã

Programarea unei interfete grafice (GUI)

Comunicarea dintre un program de aplicatie si operatorul (beneficiarul) aplicatiei poate folosi ecranul în modul text sau în modul grafic. Majoritatea aplicatiilor actuale preiau datele de la operatorul uman în mod interactiv, printr-o interfatã graficã, pusã la dispozitie de sistemul gazdã (Microsoft Windows, Linux cu X-Windows etc).

Interfata graficã cu utilizatorul (GUI = Graphical User Interface) este mai sigurã si mai "prietenoasã", folosind atât tastatura cât si mouse-ul pentru introducere sau selectare de date afisate pe ecran.

O interfatã graficã simplã constã dintr-o singurã fereastrã ecran a aplicatiei pe care se plaseazã diverse componente vizuale interactive, numite si “controale” pentru cã permit operatorului sã controleze evolutia programului prin introducerea unor date sau optiuni de lucru (care, în mod text, se transmit programului prin linia de comandã). Uneori, pe parcursul programului se deschid si alte ferestre, dar existã o fereastrã initiala cu care începe aplicatia.

Programele cu interfatã graficã sunt controlate prin evenimente produse fie de apãsarea unei taste fie de apãsarea unui buton de mouse. Un eveniment des folosit este cel produs de pozitionarea cursorului pe suprafata unui “buton” desenat pe ecran si clic pe butonul din stânga de pe mouse.

Tipul evenimentelor este determinat de componenta vizualã implicatã dar si de operatia efectuatã. De exemplu, într-un câmp cu text terminarea unei linii de text (cu “Enter”) generează un tip de eveniment, iar modificarea unor caractere din text generează un alt tip de eveniment.

Limbajul Java permite, fatã de alte limbaje, programarea mai simplã si mai versatilã a interfetei grafice prin numãrul mare de clase si de facilitãti de care dispune. De exemplu, aspectul (“Look and Feel”) componentelor vizuale poate fi ales dintre trei (patru, mai nou) variante (Windows, MacOS sau Java), indiferent de sistemul de operare gazdã.

In termenii specifici Java, componentele vizuale sunt de douã categorii:

- Componente “atomice”, folosite ca atare si care nu pot contine alte componente (un buton este un exemplu de componentã atomicã);
- Componente “container”, care grupeazã mai multe componente atomice si/sau containere.

Componentele container sunt si ele de douã feluri:

- Containere de nivel superior (“top-level”) pentru fereastra principalã a aplicatiei;
- Containere intermediare (panouri), incluse în alte containere si care permit operatii cu un grup de componente vizuale (de exemplu, pozitionarea întregului grup).

Componentele atomice pot fi grupate dupã rolul pe care îl au :

- Butoane de diverse tipuri: butoane simple, butoane radio
- Elemente de dialog
- Componente pentru selectarea unei alternative (optiuni)
- Indicatoare de progres a unor activitãti de duratã
- Componente cu text de diverse complexitati: câmp text, zonã text, documente

- Panouri cu derulare verticalã sau orizontalã (pentru liste sau texte voluminoase)
- Meniuri si bare de instrumente

In POO fiecare componentã a unei interfete grafice este un obiect dintr-o clasã predefinitã sau dintr-o subclasã a clasei de bibliotecã. Colectia claselor GUI constituie un cadru pentru dezvoltarea de aplicatii cu interfatã graficã, în sensul cã asigurã o bazã de clase esentiale si impune un anumit mod de proiectare a acestor aplicatii si de folosire a claselor existente. Acest cadru ("Framework") mai este numit si infrastructurã sau colectie de clase de bazã ("Foundation Classes").

Ca infrastructurã pentru aplicatiile Java cu interfatã graficã vom considera clasele JFC (Java Foundation Classes), numite si "Swing", care reprezintã o evolutie fatã de vechile clase AWT (Abstract Window Toolkit). Multe din clasele JFC extind sau folosesc clase AWT. Clasele JFC asigurã elementele necesare proiectãrii de interfete grafice complexe, atrãgãtoare si personalizate dupã cerintele aplicatiei si ale beneficiarilor, reducând substantial efortul de programare a unor astfel de aplicatii (inclusiv editoare de texte, navigatoare pe retea si alte utilitare folosite frecvent).

O parte din clasele JFC sunt folosite ca atare (prin instantiere) iar altele asigurã doar o bazã pentru definirea de clase derivate.

Clase JFC pentru interfatã graficã

O altã clasificare posibilã a claselor Swing este urmãtoarea:

- Clase JFC care au corespondent în clasele AWT, având aproape acelasi nume (cu prefixul 'J' la clasele JFC) si acelasi mod de utilizare: *JComponent, JButton, JCheckBox, JRadioButton, JMenu, JComboBox, JLabel, JList, JMenuBar, JPanel, JPopUpMenu, JScrollBar, JScrollPane, JTextField, JTextArea*.
- Clase JFC noi sau extinse: *JSlider, JSplitPanel, JTabbedPane, JTable, JToolBar, JTree, JProgressBar, JInternalFrame, JFileChooser, JColorChooser* etc.
- Clase de tip "model", concepute conform arhitecturii MVC ("Model-View-Controller"): *DefaultButtonModel, DefaultListSelectionModel, DefaultTreeModel, AbstractTableModel* etc.

Clasele JFC container de nivel superior sunt numai trei: *JFrame, JDialog* si *JApplet*. Primele douã sunt subclase (indirecte) ale clasei *Window* din AWT.

Toate celelalte clase JFC sunt subclase directe sau indirecte ale clasei *JComponent*, inclusiv clasa container intermediar *JPanel*.

Controalele JFC pot fi inscriptionate cu text si/sau cu imagini (încãrcate din fisiere GIF sau definite ca siruri de constante în program).

In jurul componentelor pot fi desenate borduri , fie pentru delimitarea lor, fie pentru crearea de spatii controlabile între componente vecine.

Un program minimal cu clase JFC, creeazã si afiseazã componentele vizuale pe ecran, fãrã sã trateze evenimentele asociate acestor componente. Cea mai mare parte dintr-un astfel de program creeazã în memorie structurile de date ce contin atributele componentelor vizuale si relatiile dintre ele: se creeazã un obiect fereastrã (panou), care constituie fundalul pentru celelalte componente; se creeazã componente atomice si se adaugã la panou obiectele grafice create de programator (cu metoda "add").

In final, se stabilesc dimensiunile ferestrei principale (metoda "setSize" sau

“pack”) si se comandã afisarea ferestrei principale (metoda “setVisible” sau “show”).

Fereastra principalã a aplicatiei este în general de tipul *JFrame* si contine o barã de titlu si trei “butoane” standard în coltul dreapta-sus al ferestrei pentru operatii de micsorare (minimizare), mãrire (maximizare) si închidere fereastrã .

Exemplul urmãtor afiseazã o fereastrã cu titlu, dar fãrã alte componente vizuale :

```
import javax.swing.*;
class EmptyFrame {
  public static void main ( String args[]) {
     JFrame frm = new JFrame(“EmptyFrame”);
     frm.pack();                                    // sau  frm.setSize(500,300)
     frm.setVisible (true);                         // sau  frm.show();
  }
}
```

Fereastra principalã se afiseazã initial într-o formã redusã la bara de titlu cu cele 3 butoane generale, dupã care poate fi mãritã. Pentru afisarea continutului ferestrei chiar de la început se poate folosi metoda “pack” (din clasa *JFrame*) sau metoda “setSize” (din clasa *Container*).

Adãugarea de componente vizuale la fereastra principalã *JFrame* se poate face în douã moduri. In exemplul urmãtor se extrage "panoul" ferestrei cu "getContentPane" si se adaugã o componentã *JLabel* (o etichetã) la acest panou:

```
   // adaugare la panoul extras din obiectul JFrame
import javax.swing.*;
class LabelFrame {                                  // fereastra cu o eticheta in ea
  public static void main (String args[ ]){
       JFrame frame = new JFrame();                 // fereastra aplicatiei
       JLabel label = new JLabel ("Eticheta");      // creare eticheta
       frame.getContentPane().add(label);           // adauga eticheta la fereastrã
       frame.setSize(200,200);                      // dimensiuni fereastra
       frame.setVisible(true);                      // afisare continut fereastrã
  }
}
```

In exemplul urmãtor se creeazã un panou *JPanel*, pe care se plaseazã eticheta, iar panoul este adãugat ulterior la fereastra *JFrame*:

```
     // adaugare la un panou introdus apoi in obiectul JFrame
import javax.swing.*;
class LabelFrame {                                  // fereastra cu o eticheta in ea
  public static void main (String args[]){
       JFrame frame = new JFrame();                 // fereastra aplicatiei
       JPanel panel = new JPanel();                 // un obiect “panou”
        panel.add (new JLabel ("Eticheta")) ;       // adaugã eticheta la panou
       frame.setContentPane (panel);                // sau:  frame.add (panel);
       frame.show ();                               // afisare continut fereastrã
  }
```

}

Efectuarea unui clic pe butonul de închidere al ferestrei principale (X) are ca efect închiderea ferestrei, dar nu se terminã aplicatia dacã nu estre tratat evenimentul produs de acest clic. De aceea este necesarã tratarea acestui eveniment, sau terminarea programului de cãtre operator, prin Ctrl-C. Incepând cu versiunea 1.3 mai existã o posibilitate de terminare a aplicatiei la închiderea ferestrei principale:

```
frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
```

Solutii de programare a interfetelor grafice

In general, crearea si afisarea unei interfete grafice necesitã urmãtoarele operatii din partea programatorului aplicatiei:

- Crearea unui obiect fereastrã principalã, de tip *JFrame* sau de un subtip al tipului *JFrame*, si stabilirea proprietãtilor ferestrei (titlu, culoare, dimensiuni etc.)
- Crearea componentelor atomice si stabilirea proprietãtilor acestora (dimensiuni, text afisat, culoare, tip chenar etc.)
- Gruparea componentelor atomice în containere intermediare, care sunt obiecte de tip *JPanel* sau de un subtip al acestei clase.
- Adãugarea containerelor intermediare la fereastra aplicatiei si stabilirea modului de asezare a acestora, dacã nu se preferã modul implicit de dispunere în fereastrã.
- Tratarea evenimentelor asociate componentelor si ferestrei principale, prin definirea de clase de tip “ascultãtor” la evenimentele generate de componentele vizuale.
- Afisarea ferestrei principale, prin metoda “setVisible” sau “show” a clasei *JFrame*.

Exemplele anterioare nu reprezintã solutia recomandatã pentru programarea unei interfete grafice din urmãtoarele motive:

- Variabilele referintã la obiecte JFC nu vor fi locale metodei “main” pentru cã ele sunt folosite si de alte metode, inclusiv metode activate prin evenimente.
- Metoda staticã “main” trebuie redusã la crearea unui obiect si, eventual, la apelarea unei metode pentru acel obiect. Obiectul apartine unei clase definite de programator si care foloseste clasa *JFrame* sau *JPanel*.

Vom prezenta în continuare trei variante uzuale de definire a clasei GUI în cazul simplu al unui câmp text însotit de o etichetã ce descrie continutul câmpului text.

Prima variantã foloseste o subclasã a clasei *JFrame*:

```
import javax.swing.*;
import java.awt.*;
class GUI1 extends JFrame {
  private JLabel lbl1 = new JLabel ("Directory");
  private JTextField txt1 = new JTextField (16);
   // constructor
  public GUI1 ( String title) {
    super(title);
    init();
  }
```

```
   // initializare componente
  private void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add (lbl1); cp.add(txt1);
    setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
//    txt1.addActionListener (new TxtListener());
    setSize (300,100);
  }
    // activare interfata grafica
  public static void main (String arg[])  {
    new GUI1("GUI solution 1").show();
  }
}
```

Varianta a doua foloseste “delegarea” sarcinilor legate de afisare cãtre un obiect *JFrame*, continut de clasa GUI:

```
import javax.swing.*;
import java.awt.*;
class GUI2 {
  private JFrame frame;
  private JLabel lbl1 = new JLabel ("Directory");
  private JTextField txt1 = new JTextField (16);
   // constructor
  public GUI2 ( String title) {
    frame = new JFrame(title);
    init();
    frame.show();
  }
   // initializare componente
  private void init() {
    Container cp = frame.getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add (lbl1); cp.add(txt1);
    frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
//    txt1.addActionListener (new TxtListener());
    frame.setSize (300,100);
  }
    // activare interfata grafica
  public static void main (String arg[])  {
    new GUI2("GUI solution 2");
  }
}
```

Varianta 3 defineste clasa GUI ca o subclasã a clasei *JPanel*:

```
import javax.swing.*;
class GUI3 extends JPanel {
```

```
  private JLabel lbl1 = new JLabel ("Directory");
  private JTextField txt1 = new JTextField (16);
   // constructor
  public GUI3 () {
    init();
  }
   // initializare componente
  private void init() {
    add (lbl1);
    add(txt1);
//    txt1.addActionListener (new TxtListener());
  }
    // activare interfata grafica
  public static void main (String arg[])  {
    JFrame frame = new JFrame ("GUI solution 3");
//    frame.setContentPane(new GUI3());
    frame.getContentPane().add (new GUI3());
    frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
    frame.setSize(300,100); frame.show();
  }
}
```

Clasele ascultãtor la evenimente (“TxtListener” si altele) sunt de obicei clase incluse în clasa GUI pentru a avea acces la variabilele ce definesc obiecte JFC. Dacã sunt putini ascultãtori, atunci clasa GUI poate implementa una sau câteva interfete de tip ascultator la evenimente si sã continã metodele de tratare a evenimentelor (dispare necesitatea unor clase ascultãtor separate).

Variabilele de tipuri JFC (JLabel, JTextField, s.a) pot fi initializate la declarare sau în constructorul clasei GUI, deoarece va exista un singur obiect GUI. Metoda “init” de initializare a componentelor JFC poate lipsi dacã are numai câteva linii. De observat cã pentru clasele GUI constructorul este cea mai importantã functie si uneori singura functie din clasã.

Dupã executia metodei “show” (sau “setVisible”) nu se vor mai crea alte obiecte JFC (de exemplu, ca rãspuns la evenimente generate de operatorul uman), deoarece ele nu vor mai fi vizibile pe ecran. In schimb, se practicã modificarea continutului afisat în componentele deja existente; de exemplu, metoda “setText” din clasele *JTextField* si *JLabel* poate modifica textul afisat în astfel de componente JFC.

Dispunerea componentelor într-un panou

Plasarea componentelor grafice pe un panou se poate face si prin pozitionare în coordonate absolute de cãtre programator, dar este mult mai simplu sã apelãm la un obiect de control al asezãrii în panou (“Layout Manager”), obiect selectat prin metoda “setLayout” si care stabileste automat dimensiunile si pozitia fiecãrei componente într-un panou. Pentru panoul de “continut” al ferestrei *JFrame* este activ implicit “BorderLayout”, mod care foloseste un al doilea parametru în metoda “add”. Dacã nu se specificã pozitia la adãugare, atunci componenta este centratã în fereastrã, iar dacã

sunt mai multe componente, atunci ele sunt suprapuse pe centrul ferestrei. Exemplu de plasare a trei butoane:

```
class DefaultLayout {                          // exemplu de asezare butoane in fereastra
 public static void main (String args[ ]) {
     JFrame frame = new JFrame();
     Container cp = frame.getContentPane();
     cp.add(new JButton(“ 1 ”),BorderLayout.EAST);          //  cp.add (...,”East”)
     cp.add(new JButton(“ 2 ”),BorderLayout.CENTER);     // cp.add (...,”Center”)
     cp.add(new JButton(“ 3 ”), BorderLayout.WEST);         //  cp.add (..., “West”)
     frame.setVisible(true);
 }
}
```

De observat cã obiectul extras cu "getContentPane" are tipul *Container*.

O solutie mai simplã este alegerea modului de dispunere *FlowLayout*, care aseazã componentele una dupã alta de la stânga la dreapta si de sus în jos în functie de dimensiunile componentelor si ale ferestrei principale. Exemplu:

```
class FlowLayoutDemo {                              // Asezare butoane in fereastra
  public static void main (String args[ ]) {
     JFrame frame = new JFrame();
     Container cp = frame.getContentPane();
     cp.setLayout(new FlowLayout());                // sau  new GridLayout()
     for (int i=1;i<7;i++)                          // 6 butoane cu cifre
        cp.add (new JButton (String.valueOf(i)) );
     frame.setSize(400,200);
     frame.setVisible(true);
  }
}
```

De observat cã pentru un panou *JPanel* asezarea implicitã este *FlowLayout*.

Alte modalitãti de dispunere a componentelor într-un panou sunt *GridLayout* (o matrice de componente egale ca dimensiune), *GridBagLayout*, *BoxLayout* (asezare compactã pe verticalã sau pe orizontalã, la alegere) si *CardLayout* (componente / panouri care ocupã alternativ acelasi spatiu pe ecran).

Alegerea modului de dispunere depinde de specificul aplicatiei :

- In cazul unei singure componente în panou care sã foloseascã la maximum suprafata acestuia se va alege *GridBagLayout* sau *BorderLayout* : pentru o zonã text sau o listã de selectie *JList*, de exemplu.
- In cazul a câteva componente ce trebuie sã aparã în mãrimea lor naturalã si cât mai compact se va folosi *BoxLayout* sau *FlowLayout*: pentru câteva butoane sau câmpuri text, de exemplu.
- In cazul mai multor componente de aceeasi mãrime se va folosi *GridBagLayout*: grupuri de butoane, de exemplu.
- In cazul unor componente de diferite dimensiuni *BoxLayout* sau *GridBagLayout* permite un control mai bun al plasãrii componentelor si al intervalelor dintre ele.

Asezarea componentelor într-un panou se poate modifica automat atunci când se

modificã dimensiunile panoului, dimensiunile sau numãrul componentelor (în modul *FlowLayout*). Existã diverse metode de a mentine pozitia relativã a douã sau mai multe componente vizuale, indiferent de dimensiunile panoului unde sunt plasate.

De exemplu, o etichetã (obiect *JLabel*) trebuie sã aparã întotdeauna la stânga unui câmp text sau deasupra unei zone text. Acest efect se poate obtine folosind un *GridLayout* cu douã coloane sau un *BoxLayout* cu asezare pe orizontalã.

Componente vizuale cu text

Un text scurt (nu mai lung de o linie) poate fi afisat în mod grafic (într-o zonã de pe ecran) folosind diverse componente vizuale: o etichetã (obiect *JLabel*) contine un text constant (nemodificabil din exterior), iar un câmp text (obiect *JTextField*) contine un text si permite modificarea textului de cãtre operator sau prin program.

Un câmp text (*JTextField*) permite afisarea, introducerea si editarea unei linii de text în cadrul unei ferestre text. La construirea unui obiect *JTextField* se foloseste ca parametru un sir de caractere sau un întreg pentru dimensiunea ferestrei text. Cele mai importante metode ale clasei *JTextField* sunt :

```
setText(String txt)          // afisare text din program
getText()                    // citire text introdus de utilizator in câmp
setEditable(boolean b)       // permite sau interzice editarea de text
```

In exemplul urmãtor se foloseste un câmp text (nemodificabil de la tastaturã) pentru afisarea numelui directorului curent:

```
    // afisare nume director curent
class UseTextField {
  static JFrame frame = new JFrame();
  static JTextField tf = new JTextField (20);
  public static void main (String args[]) {
    JPanel pane = (JPanel) frame.getContentPane();
    pane.setLayout( new FlowLayout());
    File dir =new File(".");                          // directorul curent
    pane.add (new JLabel("Current Directory"));       // adaugã etichetã pe panou
    tf.setText ( dir.getAbsolutePath() );             // nume cu cale completa
    tf.setEditable(false);                            // interzice editare camp text
    pane.add(tf);                                     // adaugã câmp text pe panou
    frame.pack(); frame.setVisible(true);             // comandã afisarea
  }
}
```

Principala utilizare a unui câmp text este pentru introducerea de date de cãtre operatorul aplicatiei. Pentru a informa operatorul asupra semnificatiei unui câmp text se poate adãuga o etichetã fiecãrui câmp. Exemplu de utilizare câmpuri text, cu etichete asociate, într-un formular de introducere date:

```
class InputForm {
  public static void main (String arg[]) {
```

```
    JFrame f = new JFrame();
    JPanel p =new JPanel();
    p.setLayout (new FlowLayout());      // mod de asezare in panou
    p.add (new JLabel("Nume"));          // o eticheta pentru rubrica nume
    p.add (new JTextField (20));         // rubrica pentru nume
    p.add (new JLabel("Vârsta"));        // o eticheta ptr rubrica varsta
    p.add (new JTextField (8)) ;         // rubrica pentru vârstã
    f.setContentPane(p);                 // adauga panou la fereastra principala
    f.pack(); f.setVisible(true);        // comanda afisarea pe ecran
  }
}
```

Realizarea unui formular cu mai multe rubrici este în general mai complicatã decât în exemplul anterior pentru cã necesitã controlul dispunerii câmpurilor text si etichetelor asociate, folosind alte panouri si margini de separare între aceste panouri.

O zonã text (*JTextArea*) permite afisarea mai multor linii de text, care pot fi introduse sau modificate de la tastaturã sau prin program (cu metoda “append”). Dimensiunile zonei text se stabilesc la construirea unui obiectului *JTextArea*. Exemplu de utilizare zonã text :

```
  public static void main ( String args [ ]) {
    JFrame frm = new JFrame ();
    JTextArea ta = new JTextArea(10,5);        // max 10 linii cu max 5 caractere
    frm.getContentPane().add (ta);
    for (int i=1;i<21;i++)                     // adauga siruri de cifre
       ta.append (100*i+"\n");                 // fiecare numar pe o linie separata
    frm.pack();  frm.setVisible(true);
  }
```

In AWT o zonã text este automat plasatã într-o fereastrã cu derulare pe verticalã si pe orizontalã, dar în JFC componenta *JTextArea* trebuie inclusã într-un panou cu derulare (*JScrollPane*) pentru a putea aduce în fereastra vizibilã elemente ce nu pot fi vãzute din cauza dimensiunii limitate a zonei text. Exemplu cu o zonã text pentru afisarea numelor fisierelor din directorul curent:

```
public static void main (String av[ ])  {
    File dir = new File(".");                                  // directorul curent
    String files[ ] = dir.list();                              // fisiere din directorul curent
    JFrame win= new JFrame("Current Directory");               // titlul ferestrei
    JTextArea ta = new JTextArea(20,20);                       // o zona text
    for (int i=0;i<files.length;i++)
      ta.append (" "+files[i]+"\n");                           // adauga nume fisiere la zona
    win.getContentPane().add (new JScrollPane(ta));            // în panou cu derulare
    win.pack(); win.setVisible(true);                          // comanda afisarea
  }
```

Operatorul poate deplasa cursorul în cadrul zonei text si poate modifica textul.

Exemplu de afisare a unor linii de text folosind componenta vizualã *JList*:

```
public static void main (String av[ ])  {
   File dir = new File(".");
   String files[] = dir.list();
   JFrame win= new JFrame("Current Directory");
   JList list = new JList(files);
   win.getContentPane().add (new JScrollPane(list));
   win.setSize(200,400); win.setVisible(true);
 }
```

Clasa *JComboBox* permite crearea de obiecte care contin mai multe linii scurte de text, din care se poate selecta o linie. Pe ecran se afiseazã numai linia selectatã, dar prin mouse se poate cere afisarea tuturor liniilor, pentru o altã selectie. Exemplu:

```
public static void main (String arg[ ]) {
     JFrame frm = new JFrame();
     String s[ ]={"Name","Ext","Date","Size"};        // texte afisate in ComboBox
     JComboBox cbox = new JComboBox(s);       // creare obiect ComboBox
     JLabel et = new JLabel ("Sort By: ");            // o eticheta asociata
     Container cp = frm.getContentPane();
     cp.add (et,"West");  cp.add (cbox,"Center");   // adauga eticheta si ComboBox
     frm.pack(); frm.show();                          // comanda afisarea
  }
```

Pentru ca programul sã poatã reactiona la selectarea sau modificarea unor linii trebuie adãugate programelor anterioare secvente de tratare a evenimentelor produse de aceste actiuni exterioare programului. Tratarea unui eveniment se face într-o metodã cu nume si argumente impuse (functie de tipul evenimentului), metodã inclusã într-o clasã care implementeazã o anumitã interfatã (functie de eveniment). Exemplu de tratare a evenimentului de clic pe buton prin afisarea unui dialog:

```
    // clasa ascultator la buton
class BListener implements ActionListener {
   public void actionPerformed(ActionEvent e) {
      JOptionPane.showMessageDialog(new JFrame(),"","Event Fired !",
            JOptionPane.PLAIN_MESSAGE);
   }
}
  // clasa aplicatiei
class FButon extends JFrame  {
  public static void main(String av[]) {
     JButton button = new JButton("Click Me");
     FButon b = new FButon();
     b.getContentPane().add(button, “Center”);
     button.addActionListener(new BListener());   // atasare ascultator la buton
     b.setSize(100,100); b.show();
  }
}
```

Panouri multiple

In cadrul unei ferestre principale avem urmãtoarele posibilitãti de lucru cu panouri:

- Panouri multiple afisate simultan, fãrã suprapunere între ele.
- Panouri divizate în douã (pe orintalã sau pe verticalã): *JSplitPanel*.
- Panouri multiple afisate succesiv in aceeasi zonã ecran : *JTabbedPane*.
- Panouri multiple afisate simultan si partial suprapus: *JLayeredPane*.

In fiecare dintre panourile unei aplicatii putem avea un panou cu derulare *JScrollPane*.

Panourile pot fi independente unele de altele sau pot fi legate, astfel ca selectarea unui element dintr-un panou sã aibã ca efect modificarea continutului celuilalt panou.

Pentru inserarea de spatii între panouri se pot crea margini (borduri) în jurul fiecãrui panou.

Gruparea unor componente în (sub)panouri permite manipularea unui panou separat de celelalte si alegerea altui mod de dispunere în fiecare panou. Exemplu cu douã panouri independente afisate simultan

```
class MFrame extends JFrame {
    JPanel p1 = new JPanel();                    // un panou cu doua butoane
    JPanel p2 = new JPanel();                    // alt panou cu butoane
    JButton b[] = new JButton[4];                // 4 butoane
  public MFrame ()  {
     Container w = getContentPane();             // panou principal al aplicatiei
     for (int k=0;k<4;k++)
        b[k]= new JButton(“ “+ k +” “);          // creare butoane
     p1.setLayout (new FlowLayout());            // dispunere in panoul p1
     p1.add (b[0]); p1.add (b[1]);               // butoane din panoul p1
     p2.setLayout (new FlowLayout() );           // dispunere in panoul p2
     p2.add (b[2]); p2.add (b[3]);               // butoane din panoul p2
     w.add (p1,”North”); w.add (p2,”South”);     // adauga panouri la panou princ
 }
    // utilizare MFrame
  public static void main ( String args []) {
    JFrame f = new MFrame ();
    f.setSize (100,100);  f.show ();
  }
}
```

Exemplul urmãtor afiseazã în douã panouri "legate" douã liste de fisiere diferite, dar el poate fi modificat astfel ca în panoul din dreapta sã se afiseze continutul fisierului director selectat din lista afisatã în panoul din stânga .

```
class SplitPane extends JFrame {
 public SplitPane() {
   JScrollPane p1 = dirlist(".");
   JScrollPane p2 = dirlist("c:\\");
   JSplitPane sp = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT, p1, p2);
```

```
  getContentPane().add(sp);
 }
 public static JScrollPane dirlist (String dirname) {
  String files[] = new File(dirname).list();
  JList lst = new JList(files);
  return new JScrollPane(lst);
 }
 public static void main(String s[]) {
  JFrame frame = new SplitPane();
  frame.setSize(400,200);
  frame.setVisible(true);
 }
}
```

In exemplul urmãtor cele douã panouri sunt afisate alternativ, în aceeasi zonã, în functie de selectia operatorului (fiecare panou are un “tab” de prindere, adicã o micã portiune cu numele panoului, afisatã permanent pe ecran alãturi de “tab”-urile celorlalte panouri selectabile).

```
class TabbedPane extends JFrame {
 public TabbedPane() {
  String t1=".", t2="C:\\";
  JScrollPane p1 = dirlist(t1);
  JScrollPane p2 = dirlist(t2);
  JTabbedPane tabbedPane = new JTabbedPane();
  tabbedPane.addTab("Dir of "+t1, null, p1, "");
  tabbedPane.addTab("Dir of "+t2, null, p2, "");
  setLayout(new GridLayout(1, 1));
  add(tabbedPane);
 }
 ...
}
```

Apleti Java

Cuvântul aplet (“applet”) desemneazã o micã aplicatie care foloseste ecranul în mod grafic, dar care depinde de un alt program “gazdã” pentru crearea fereastrei principale (care nu trebuie creatã de programatorul apletului). Programul gazdã este fie un program navigator (“browser”), fie programul “appletviewer”, destinat vizualizãrii rezultatului executiei unui aplet. Codul unui aplet (fisierul .class) poate fi adus de cãtre browser si de la un alt calculator decât cel pe care se executã.

Din punct de vedere sintactic un aplet este o clasã Java, derivatã din clasa *Applet* sau din *JApplet*.

Clasa *JApplet* este indirect derivatã din clasa *Panel* , care asigurã oricãrui aplet o fereastrã cu butoane de închidere, mãrire si micsorare. Fereastra de afisare a unui aplet nu poate fi manipulatã direct de operatorul uman ci numai indirect, prin fereastra programului browser.

Programarea unei interfete grafice într-un aplet este putin mai simplã decât într-o

aplicatie deoarece apletul mosteneste de la clasa *Panel* (si de la clasele *Container* si *Component*) o serie de metode utile (inclusiv metoda “windowClosing”). Exemplu de aplet scris în varianta AWT:

```
import java.awt.*;
import java.applet.*;
public class LabelAplet extends Applet {
  Label et= new Label ("Eticheta",Label.CENTER);
  public void init () {
    add (et);
  }
}
```

Acelasi aplet în varianta JFC aratã astfel:

```
import javax.swing.*;
public class JAplet extends JApplet {
  JLabel et= new JLabel ("Eticheta",JLabel.CENTER);
  public void init () {
    getContentPane().add (et);
  }
}
```

Clasa *JApplet* este derivatã din clasa *Applet* si permite în plus folosirea unui meniu într-un aplet si a componentelor vizuale noi din JFC (fãrã echivalent în AWT).

Fisierul “class” generat de compilator pentru un aplet este specificat într-un fisier “html”, împreunã cu dimensiunile ferestrei folosite de aplet, între marcajele <applet> si </applet>. Exemplu de fisier “html” necesar pentru executia apletului precedent:

```
  <applet code="JAplet.class"  width="250" height="100"> </applet>
```

In comanda “appletviewer” este specificat numele fisierului “html” si nu apare direct numele fisierului “class”. Dimensiunile ferestrei folosite de aplet se dau în fisierul de tip "html" si nu în codul Java.

Este posibil ca anumite programe de navigare mai vechi (“Browser”) sã nu recunoascã clase JFC si din acest motiv s-a dat si varianta AWT pentru aplet.

De remarcat cã o clasã care corespunde unui aplet trebuie sã aibã atributul *public* si nu contine o metodã “main”. Clasa aplet mosteneste si redefineste de obicei metodele “init”, “start”, "paint" si alte câteva metode, apelate de programul gazdã la producerea anumitor evenimente.

O clasã aplet poate fi transformatã într-o aplicatie prin adãugarea unei functii “main” în care se construieste un obiect *JFrame,* la care se adaugã un obiect aplet si se apeleazã metoda “init”:

```
public static void main (String args[ ]) {        // se adauga la clasa JAplet
  JFrame f = new JFrame();
  JAplet aplet = new JAplet();
```

```
  f.getContentPane().add (aplet);
  aplet.init();
  f.setVisible (true);
}
```

O altã posibilitate este crearea unei clase separate în care se preia codul din aplet si se adaugã crearea si afisarea ferestrei principale a aplicatiei (de tip *JFrame*). Functia “init” este înlocuitã cu functia “main” la trecerea de la un aplet la o aplicatie.

Din punct de vedere functional un aplet contine câteva functii, care trebuie (re)definite de utilizator si sunt apelate de programul gazdã. Un aplet care tratează evenimente externe trebuie sã continã si metodele de tratare a evenimentelor, pentru cã nu se admit alte clase ascultãtor, separate de clasa aplet. Obiectul ascultãtor la evenimente este chiar obiectul aplet, ceea ce conduce la instructiuni de forma urmãtoare

```
comp.addXXXListener(this);       // comp este numele unei componente din aplet
```

Exemplul urmãtor este un aplet care afiseazã un buton în centrul ferestrei puse la dispozitie de programul gazdã si emite un semnal sonor ("beep") la "apãsarea" pe buton, adicã la actionarea butonului din stânga de pe mouse dupã mutare mouse pe zona ecran ocupatã de buton.

```
public class Aplet extends JApplet implements ActionListener {
  JButton button;
  public void init() {
    button = new JButton("Click Me");
    getContentPane().add(button, BorderLayout.CENTER);
    button.addActionListener(this);        // obiectul receptor este chiar apletul
  }
  public void actionPerformed(ActionEvent e) {        // tratare eveniment buton
     Toolkit.getDefaultToolkit().beep();                    // semnal sonor
  }
}
```

Metoda "init" este apelatã o singurã datã, la încãrcarea codului apletului în memorie, iar metoda "start" este apelatã de fiecare datã când programul browser readuce pe ecran pagina html care contine si marcajul <applet ...>. Metoda "paint" are un parametru de tip *Graphics*, iar clasa *Graphics* contine metode pentru desenarea de figuri geometrice diverse si pentru afisarea de caractere cu diverse forme si mãrimi:

```
public void paint (Graphics g) {
   g.drawRect (0, 0, getSize().width - 1, getSize().height - 1);  // margini fereastra
   g.drawString ("text in aplet", 10, 30);                              // afisare text
}
```

11. Programare bazatã pe evenimente

Evenimente Swing

Programarea dirijatã de evenimente (“Event Driven Programming”) se referã la scrierea unor programe care reactioneazã la evenimente externe programului (cauzate de operatorul uman care foloseste programul). Prin “eveniment” se întelege aici un eveniment asincron, independent de evolutia programului si al cãrui moment de producere nu poate fi prevãzut la scrierea programului. Evenimente tipice sunt: apãsarea unei taste, actionarea unui buton de “mouse”, deplasare “mouse” s.a. Notiunea de eveniment a apãrut ca o abstractizare a unei întreruperi externe .

Un program controlat prin evenimente nu initiazã momentul introducerii datelor, dar poate reactiona prompt la orice eveniment produs de o actiune a operatorului. Functiile care preiau date nu sunt apelate direct si explicit de alte functii din program, ci sunt apelate ca urmare a producerii unor evenimente.

Structura unui program dirijat prin evenimente diferã de structura unui program obisnuit prin existenta functiilor speciale de tratare a evenimentelor (“Event handlers”), care nu sunt apelate direct din program. Intr-un program dirijat de evenimente existã douã tipuri principale de obiecte:

- Obiecte generatoare de evenimente (sursa unui eveniment);
- Obiecte receptoare de evenimente (obiecte ascultãtor).

Clasele generator sunt în general clase JFC sau AWT si ele creeazã obiecte “eveniment” ca efect al actiunii operatorului pe suprafata componentei respective.

Clasele receptor de evenimente sunt scrise de cãtre programatorul aplicatiei pentru cã metodele de tratare a evenimentelor observate sunt specifice fiecãrei aplicatii. Aceste clase trebuie sã implementeze anumite interfete JFC, deci trebuie sã continã anumite metode cu nume si semnãturã impuse de clasele JFC.

In Java un eveniment este un obiect de un tip clasã derivat din clasa *EventObject*.

Declansarea unui eveniment are ca efect apelarea de cãtre obiectul generator a unei metode din obiectul ascultãtor, care primeste ca argument un obiect “eveniment”. Tipul obiectului eveniment este determinat de tipul componetei GUI care a generat evenimentul dar si de actiunea operatorului uman. De exemplu, un clic pe butonul de închidere a unei ferestre *JFrame* genereazã un alt eveniment decât un clic pe butonul de micsorare a ferestrei.

Evenimentele JFC pot fi clasificate astfel:

- Evenimente asociate fiecãrei componente vizuale (buton, câmp text, etc.), generate fie prin “mouse”, fie din taste (apãsare buton, tastare “Enter”, s.a.).
- Evenimente asociate dispozitivelor de introducere (mouse sau tastaturã).

La fiecare obiect generator de evenimente se pot “înregistra” (se pot înscrie) mai multe obiecte ascultãtor interesate de producerea evenimentelor generate. Operatia de înregistrare se face prin apelarea unei metode de forma “addXListener” (din obiectul generator), unde ‘X’ este numele (tipul) evenimentului si care este totodatã si numele unei interfete.

Tratarea evenimentelor Swing

Programatorul unei aplicatii Java cu evenimente trebuie:
- Sã defineascã clasele ascultãtor, care implementeazã interfete JFC, prin definirea metodei sau metodelor interfetei pentru tratarea evenimentele observate.
- Sã înregistreze obiectele ascultãtor la obiectele generatoare de evenimente.

Anumite interfete ascultãtor ("listener") contin o singurã metodã, dar alte interfete contin mai multe metode ce corespund unor evenimente diferite asociate aceleeasi componente vizuale. De exemplu, interfata *WindowListener* contine sapte metode pentru tratarea diferitelor evenimente asociate unei ferestre (deschidere, închidere, activare, dezactivare, micsorare).

Pentru a trata numai evenimentul "închidere fereastrã" este suficient sã scriem numai metoda "windowClosing". Dacã s-ar defini o clasã care sã implementeze aceastã interfatã atunci ar trebui sã definim toate cele sapte metode ale interfetei, majoritatea cu definitie nulã (fãrã efect). Pentru a simplifica sarcina programatorului este prevãzutã o clasã "adaptor" *WindowAdapter* care contine definitii cu efect nul pentru toate metodele interfetei, iar programatorul trebuie sã redefineascã doar una sau câteva metode. Metoda "windowClosing" din aceastã clasã nu are nici un efect si trebuie redefinitã pentru terminarea aplicatiei la închiderea ferestrei principale.

Pentru a compila exemplele care urmeazã se vor introduce la început urmãtoarele instructiuni:

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

In exemplul urmãtor se defineste o clasã separatã, care extinde clasa *WindowAdapter* si redefineste metoda "windowClosing":

```
class WindExit extends WindowAdapter {
     public void windowClosing( WindowEvent e) {
        System.exit(0);
     }
}
class FrameExit {
  public static void main (String args[ ]) {
     JFrame f = new JFrame ();
     f.addWindowListener ( new WindExit());
     f.show ();
  }
}
```

Putem folosi si o clasã inclusã anonimã pentru obiectul ascultãtor necesar:

```
public static void main (String args[ ]) {
     JFrame f = new JFrame ();
```

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

Evenimente de mouse si de tastaturã

Interfata *KeyListener* contine trei metode : “keyPressed”, “keyTyped” si “keyReleased”, iar clasa *KeyAdapter* contine definitii nule pentru cele trei metode. In exemplul urmãtor este tratat numai evenimentul de tastã apãsatã prin afisarea caracterului corespunzãtor tastei:

```
class KeyEvent1 {
   static JFrame f= new JFrame();
   static JTextArea ta; static JTextField tf;
   class KeyHandler extends KeyAdapter {
     public void keyPressed(KeyEvent e) {
       ta.append("\n key typed: " + e.getKeyChar() );
     }
   }
   public static void main(String args[]) {
     KeyEvent1 kd = new KeyEvent1();
     tf = new JTextField(20);                    // aici se introduc caractere
     tf.addKeyListener(kd.new KeyHandler());
     ta = new JTextArea(20,20);                  // aici se afiseaza caracterele introduse
     Container cp = f.getContentPane();
     cp.setLayout(new GridLayout());
     cp.add(tf);  cp.add(new JScrollPane (ta));
     f.show( );
   }
```

Evenimentele de la tastaturã sunt asociate componentei selectate prin mouse. Este posibilã si selectarea prin program a componentei pe care se focalizeazã tastatura folosind metoda care existã în orice componentã JFC numitã “requestFocus”. In exemplul urmãtor s-a adãugat un buton de stergere a continutului zonei text, a cãrui apãsare are ca efect focalizarea tastaturii pe câmpul text.

```
class KeyEvent2 {
   static JFrame f= new JFrame();
   static JTextArea ta;  static JTextField tf;
   static JButton button = new JButton("Clear");
   class KeyHandler extends KeyAdapter {          // tratare eveniment tastatura
     public void keyPressed(KeyEvent e) {
            ta.append("\n key typed: " + e.getKeyChar() );
     }
   }
```

```
        // tratare eveniment buton
    class ActHandler implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        ta.setText("");  tf.setText("");
        tf.requestFocus();                  // refocalizare pe TextField
      }
    }
   . . .
}
```

Programele anterioare lucrează corect numai pentru taste cu caractere afisabile, dar ele pot fi extinse pentru a prezenta orice caracter introdus.

Interfata *MouseInputListener* contine metode apelate la actiuni pe mouse (apãsarea sau eliberare buton: “mouseClicked”, “mousePressed”, “mouseReleased”) si metode apelate la deplasare mouse (“mouseMoved”,”mouseDragged”). *MouseInputAdapter* oferã o implementare nulã pentru toate cele 7 metode. Exemplul urmãtor aratã cum se poate reactiona la evenimentul de “clic” pe mouse prin numãrarea acestor evenimente si afisarea lor.

```
class MouseClicks {
  static int clicks=0;                                  // numar de apasari pe mouse
  public static void main (String args[]) {
     JFrame frame = new JFrame();
     final JLabel label= new JLabel("0");                  // o comp. neinteractiva
     label.addMouseListener (new MouseInputAdapter() {    // receptor evenimente
     public void mouseClicked(MouseEvent e) {
       ++clicks;
       label.setText((new Integer(clicks)).toString());            // modifica afisarea
     }
  });
  frame.getContentPane().setLayout (new FlowLayout());
  frame.getContentPane().add(label);
  frame.setVisible(true);
  }
}
```

Evenimente asociate componentelor JFC

Orice componentã vizualã AWT sau JFC poate fi sursa unui eveniment sau chiar a mai multor tipuri de evenimente, cauzate de un clic pe imaginea componentei sau de apãsarea unei taste.

Majoritatea componentelor vizuale sunt interactive, în sensul cã dupã ce se afiseazã forma si continutul componentei este posibil un clic cu mouse-ul pe suprafata componentei sau deplasarea unor elemente ale componentei, ceea ce generează evenimente. De asemenea editarea textului dintr-un câmp text sau dintr-o zonã text produce evenimente.

Fiecare componentã generează anumite tipuri de evenimente, si pentru fiecare tip de eveniment este prevãzutã o metodã de tratare a evenimentului. Metodele de tratare

sunt grupate în interfete. Programatorul trebuie sã defineascã clase ce implementeazã aceste interfete cu metode de tratare a evenimentelor. Astfel de clase “adaptor” au rolul de intermediar între componentã si aplicatie.

Evenimentele generate de componentele JFC pot fi clasificate în:

- Evenimente comune tuturor componentelor JFC:

ActionEvent, FocusEvent, KeyEvent, MouseEvent, ComponentEvent

- Evenimente specifice fiecãrui tip de componentã:

ChangeEvent, MenuEvent, ListDataEvent, ListSelectionEvent, DocumentEvent etc.

Evenimentele comune, mostenite de la clasa *Component*, pot fi descrise astfel:

- *ActionEvent* : produs de actiunea asupra unei componente prin clic pe mouse.
- *ComponentEvent* : produs de o modificare în dimensiunea, pozitia sau vizibilitatea componentei.
- *FocusEvent*: produs de “focalizarea” claviaturii pe o anumitã componentã, pentru ca ea sã poatã primi intrãri de la tastaturã.
- *KeyEvent* : produs de apãsarea unei taste si asociat componentei pe care este focalizatã tastatura.
- *MouseEvent* : produs de apãsare sau deplasare mouse pe suprafata componentei.

Numele evenimentelor apar în clasele pentru obiecte “eveniment” si în numele unor metode: “add*Listener”, “remove*Listener”.

Un buton este un obiect Swing de tip *JButton* care generează câteva tipuri de evenimente: *ActionEvent* (dacã s-a actionat asupra butonului), *ChangeEvent* (dacã s-a modificat ceva în starea butonului) s.a. La apãsarea unui buton se creează un obiect de tip *ActionEvent* si se apeleazã metoda numitã “actionPerformed” (din interfata *ActionListener*) pentru obiectul sau obiectele înregistrare ca receptori la acel buton (prin apelul metodei “addActionListener”). Tratarea evenimentului înseamnã scrierea unei metode cu numele “actionPerformed” care sã producã un anumit efect ca urmare a “apãsãrii” butonului (prin clic pe suprafata sa).

In general nu se tratează toate evenimentele ce pot fi generate de o componentã JFC. Desi un buton poate genera peste 5 tipuri de evenimente, în mod uzual se foloseste numai *ActionEvent* si deci se apeleazã numai metoda “actionPerformed” din obiectele înregistrate ca receptori pentru butonul respectiv. Se poate spune cã se produc efectiv numai evenimentele pentru care s-au definit ascultãtori si deci metode de tratare a evenimentelor. Exemplu de tratare a evenimentului de clic pe un buton, prin emiterea unui semnal sonor la fiecare clic:

```
    // clasa ptr obiecte ascultator de evenimente
class Listener implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    Toolkit.getDefaultToolkit().beep();                 // produce semnal sonor (beep)
  }
}
    // creare buton si asociere cu ascultator
class Beeper1 {
  public static void main ( String args[]) {
    JFrame frame = new JFrame();
    JButton buton = new JButton("Click Me");
```

```
    frame.getContentPane().add(buton, BorderLayout.CENTER);
    button.addActionListener(new Listener());       // inscriere receptor la buton
    frame.setVisible(true);
    }
}
```

Pentru a obtine acelasi efect si prin apãsarea unei taste asociate butonului (de exemplu combinatia de taste Ctrl-C) este suficient sã adãugãm o instructiune:

```
    buton.setMnemonic (KeyEvent.VK_C);
```

Se observã cã am folosit clasa “Listener” o singurã datã, pentru a crea un singur obiect. De aceea se practicã frecvent definirea unei clase “ascultãtor” anonime acolo unde este necesarã:

```
button.addActionListener(new ActionListener(){       // definitia clasei receptor
  public void actionPerformed(ActionEvent e){
    Toolkit.getDefaultToolkit().beep();
  }
});
```

Diferenta dintre evenimentul generat de un buton si un eveniment generat de clic pe mouse este cã primul apare numai dacã dispozitivul mouse este pozitionat pe aria ocupatã de buton, în timp ce al doilea apare indiferent de pozitia mouse pe ecran. In plus, evenimentele generate de componente JFC contin în ele sursa evenimentului si alte informatii specifice fiecãrei componente.

Exemplul urmãtor foloseste componente de tip *JCheckBox* (cãsute cu bifare) care genereazã alt tip de evenimente (*ItemEvent*), ilustreazã un ascultãtor la mai multe surse de evenimente si utilizarea unei metode “removeXXXListener”:

```
class CheckBox1 extends JFrame {
   JCheckBox cb[] = new JCheckBox[3];
   JTextField a = new JTextField(30);
   String [ ] aa = {"A","B","C"};                    // litere afisate in casute
   public CheckBox1() {
     CheckBoxListener myListener = new CheckBoxListener();// ascultator  casute
      for (int i=0;i<3;i++) {
         cb[i] = new JCheckBox (aa[i]);              // creare 3 casute
         cb[i].addItemListener (myListener);         // acelasi ascultator la toate casutele
      }
      Container cp = getContentPane();
       cp.setLayout(new GridLayout(0, 1));
        for (int i=0;i<3;i++)
           cp.add(cb[i]);
        cp.add(a);
   }
      // Ascultator la casute cu bifare.
   class CheckBoxListener implements ItemListener {
      public void itemStateChanged(ItemEvent e) {
```

```
        JCheckBox source = (JCheckBox)e.getSource();
        for (int i=0;i<3;i++)
          if (source == cb[i]) {
            source.removeItemListener(this);        // ptr a evita repetarea selectiei
           a.setText(a.getText()+aa[i]);            // afisare nume casutã selectata
          }
       }
    }
```

Evenimente produse de componente text

Pentru preluarea caracterelor introduse într-un câmp text trebuie tratat evenimentul *ActionEvent*, generat de un obiect *JTextField* la apãsarea tastei "Enter". Pânã la apãsarea tastei “Enter” este posibilã si editarea (corectarea) textului introdus.

Exemplu de validare a continutului unui câmp text la terminarea introducerii, cu emiterea unui semnal sonor în caz de eroare :

```
class MFrame extends JFrame {
  JTextField tf;
  class TFListener implements ActionListener {          // clasa interioara
    public void actionPerformed (ActionEvent ev) {
      String text = tf.getText();                       // text introdus de operator
      if ( ! isNumber(text))                            // daca nu sunt doar cifre
        Toolkit.getDefaultToolkit().beep();             // emite semnal sonor
    }
  }
  public MFrame ()  {
    Container w = getContentPane();
    tf = new JTextField(10);
    w.add (tf,"Center");
    tf.addActionListener (new TFListener());
 }
   // verifica daca un sir contine numai cifre
 static boolean isNumber (String str) {
    for (int i=0;i<str.length();i++)
      if ( ! Character.isDigit(str.charAt(i)) )
        return false;
    return true;
  }
 public static void main ( String av[]) {
   JFrame f = new MFrame ();
   f.pack();  f.setVisible(true);
 }
}
```

Unele aplicatii preferã sã adauge un buton care sã declanseze actiunea de utilizare a textului introdus în câmpul text, în locul tastei “Enter” sau ca o alternativã posibilã.

Selectarea unui text dintr-o listã de optiuni *JComboBox* produce tot un eveniment de tip *ActionEvent*. Exemplu:

```
    // ascultator la ComboBox
class CBListener implements ActionListener {
    JTextField tf;
    public CBListener (JTextField tf) {
      this.tf=tf;
    }
    public void actionPerformed (ActionEvent ev) {
      JComboBox cb = (JComboBox) ev.getSource();        // sursa eveniment
      String seltxt = (String) (cb.getSelectedItem());       // text selectat
      tf.setText (seltxt);                                   // afisare in campul text
    }
}
class A {
  public static void main (String arg[]) {
     JFrame frm = new JFrame();
     String s[]={"Unu","Doi","Trei","Patru","Cinci"};     // lista de optiuni
     JComboBox cbox = new JComboBox(s);
     JTextField txt = new JTextField(10);                      // ptr afisare rezultat selectie
     Container cp = frm.getContentPane();
     cp.add (txt,"South");  cp.add (cbox,"Center");
     cbox.addActionListener ( new CBListener (txt));      // ascultator la ComboBox
     frm.pack(); frm.show();
  }
}
```

Mecanismul de generare a evenimentelor

De cele mai multe ori, în Java, folosim surse de evenimente prefabricate (componente AWT sau JFC) dar uneori trebuie să scriem clase generatoare de evenimente, folosind alte clase si interfete JDK. Acest lucru este necesar atunci când trebuie creată o componentă standard Java (“bean”), deoarece evenimentele fac parte din standardul pentru JavaBeans (componentele standard comunică între ele si prin evenimente, chiar dacă nu sunt componente “vizuale”).

La o sursă de evenimente se pot conecta mai multi “ascultători” interesati de aceste evenimente, după cum este posibil ca un acelasi obiect să poată “asculta” evenimente produse de mai multe surse.

O sursă de evenimente trebuie să conțină o listă de receptori ai evenimentelor generate, care era un vector de referinte la obiecte de tipul *Object* (in JDK 1.3 s-a introdus clasa *EventListenerList* pentru a manipula mai sigur această listă).

Un eveniment este încapsulat într-un obiect de un tip subclasă a clasei generale *EventObject*, definite în pachetul “java.util” astfel:

```
    public class EventObject extends Object implements Serializable {
        public EventObject (Object source);        // primeste sursa evenimentului
        public Object getSource();                 // produce sursa evenimentului
        public String toString();                  // un sir echivalent obiectului
    }
```

Mecanismul de producere a evenimentelor si de notificare a obiectelor “ascultãtor” poate fi urmãrit în cadrul claselor JFC de tip “model”, unde cuvântul “model” are sensul de model logic al datelor care vor fi prezentate vizual. O clasã model contine date si poate genera evenimente la modificarea sau selectarea unor date continute.

Modelul de buton (*DefaultButtonModel*) este o sursã de evenimente si contine o listã de “ascultãtori” sub forma unui obiect de tipul *EventListenerList*, care este în esentã un vector de referinte la obiecte ce implementeazã interfata *EventListener*. Inregistrarea unui obiect ascultãtor la obiectul sursã de evenimente se face prin apelarea metodei “addActionListener”.

Un model de buton trebuie sã respecte interfata *ButtonModel*, redatã mai jos într-o formã mult simplificatã (prin eliminarea a cca 70 % dintre metode):

```
public interface ButtonModel {
   boolean isPressed();
   public void setPressed(boolean b);
   public void setActionCommand(String s);
   public String getActionCommand();
   void addActionListener(ActionListener l);
   void removeActionListener(ActionListener l);
}
```

Sirul numit “ActionCommand” reprezintã inscriptia afisatã pe buton si permite identificarea fiecãrui buton prin program (este diferit de numele variabilei *JButton*).

Secventa urmãtoare contine o selectie de fragmente din clasa model de buton, considerate semnificative pentru aceastã discutie:

```
public class DefaultButtonModel implements ButtonModel, Serializable {
   protected int stateMask = 0;
   protected String actionCommand = null;
   public final static int PRESSED = 1 << 2;
   protected transient ChangeEvent changeEvent = null;
        // lista de ascultatori la evenimente generate de aceasta clasa
   protected EventListenerList listenerList = new EventListenerList();
        // actionare buton prin program
   public void setPressed(boolean b) {
      if((isPressed() == b) || !isEnabled()) {  return;  }
      if (b)   stateMask |= PRESSED;
      else     stateMask &= ~PRESSED;
         // declansare eveniment “buton actionat”
      if(!isPressed() && isArmed()) {
         fireActionPerformed(
            new ActionEvent (this , ActionEvent.ACTION_PERFORMED,
                  getActionCommand()) );
      }
      fireStateChanged();
   }
        // adaugare receptor la evenimente generate de buton
```

```
    public void addActionListener(ActionListener l) {
        listenerList.add(ActionListener.class, l);
    }
        // eliminare receptor la evenimente buton
    public void removeActionListener(ActionListener l) {
        listenerList.remove(ActionListener.class, l);
    }
          // notificare receptori despre producere eveniment
    protected void fireActionPerformed (ActionEvent e) {
        Object[] listeners = listenerList.getListenerList();
        for (int i = listeners.length-2; i>=0; i-=2)
            if (listeners[i]==ActionListener.class)
                ((ActionListener)listeners[i+1]).actionPerformed(e);
    }
          // notificare receptori despre schimbarea stării
       protected void fireStateChanged() {
            Object[ ] listeners = listenerList.getListenerList();
           for (int i = listeners.length-2; i>=0; i-=2) {
            if (listeners[i]==ChangeListener.class) {
                if (changeEvent == null)
                    changeEvent = new ChangeEvent(this);
                ((ChangeListener)listeners[i+1]).stateChanged(changeEvent);
            }
        }
    }
}
```

Apelarea metodei “setPressed” pentru un buton are acelasi efect cu actiunea operatorului de clic pe imaginea butonului, adicã declanseazã un eveniment de tip “ActionEvent” si deci apeleazã metodele “actionPerformed” din toate obiectele ascultator înregistrate la butonul respectiv.

Existã un singur obiect eveniment *ChangeEvent* transmis ascultãtorilor la orice clic pe buton, dar câte un nou obiect *ActionEvent* pentru fiecare clic, deoarece sirul “actionCommand” poate fi modificat prin apelul metodei “setActionCommand”.

Producerea unui eveniment *ActionEvent* se traduce în apelarea metodei “actionPerformed” pentru toate obiectele de tip *ActionListener* înregistrate.

Lista de ascultãtori la diferite evenimente este unicã pentru un obiect *EventListenerList*, dar în listã se memoreazã si tipul (clasa) obiectului ascultãtor, împreunã cu adresa sa. Intr-o formã mult simplificatã, clasa pentru liste de obiecte ascultãtor aratã astfel:

```
public class EventListenerList  {
  protected transient Object[ ] listenerList = null;   // lista de obiecte ascultator
    // adauga un obiect ascultator la lista
  public synchronized void add (Class t, EventListener l) {
      if (listenerList == null)
          listenerList = new Object[ ] { t, l };
      else {
          int i = listenerList.length;
         Object[ ] tmp = new Object[i+2];         // realocare pentru extindere vector
```

```
        System.arraycopy(listenerList, 0, tmp, 0, i); // copiere date in "tmp"
         tmp[i] = t; tmp[i+1] = l;                    // adaugare la vectorul "tmp"
         listenerList = tmp;
      }
  }
}
```

O solutie mai simplã dar mai putin performantã pentru clasa *EventListenerList* poate folosi o colectie sincronizatã în locul unui vector intrinsec extins mereu.

Structura programelor dirijate de evenimente

Intr-un program care reactioneazã la evenimente externe trebuie definite clase ascultãtor pentru aceste evenimente. De multe ori clasele ascultãtor trebuie sã comunice între ele, fie direct, fie prin intermediul unei alte clase. In astfel de situatii avem de ales între mai multe posibilitãti de grupare a claselor din program, fiecare cu avantaje si dezavantaje.

Pentru a ilustra variantele posibile vom folosi un exemplu simplu cu douã butoane si un câmp text. In câmpul text se afiseazã un numãr întreg (initial zero); primul buton (‘+’) are ca efect mãrirea cu 1 a numãrului afisat iar al doilea buton (‘-’) produce scãderea cu 1 a numãrului afisat. Desi foarte simplu, acest exemplu aratã necesitatea interactiunii dintre componentele vizuale si obiectele “ascultãtor”.

Prima variantã foloseste numai clase de nivel superior (“top-level”): o clasã pentru fereastra principalã a aplicatiei si douã clase pentru tratarea evenimentelor de butoane. Obiectele ascultãtor la butoanele ‘+’ si ‘-‘ trebuie sã actioneze asupra unui câmp text si deci trebuie sã primeascã o referintã la acest câmp text (în constructor).

```
    // ascultator la buton "+"
class B1L implements ActionListener {
   JTextField text;                          // referinta la campul text folosit
   public B1L (JTextField t) {  text=t;  }
   public void actionPerformed (ActionEvent ev) {
      int n =Integer.parseInt(text.getText());   // valoarea din campul text
      text.setText(String.valueOf(n+1));       // modifica continut camp text
   }
 }
    // ascultator la buton "-"
class B2L implements ActionListener {
   JTextField text;                          // referinta la campul text folosit
   public B2L (JTextField t) { text=t; }
   public void actionPerformed (ActionEvent ev) {
      int n =Integer.parseInt(text.getText());   // valoarea din campul text
      text.setText(String.valueOf(n-1));       // modifica continut camp text
   }
 }
    // Clasa aplicatiei: butoane cu efect asupra unui camp text
class MFrame extends JFrame  {
  JButton b1 = new JButton ("  +  ");
```

```
  JButton b2 = new JButton ("  -  ");
  JTextField text = new JTextField (6);
  public MFrame() {
    Container c = getContentPane();
    text.setText("0");
    b1.addActionListener (new B1L(text) );
    b2.addActionListener (new B2L(text) );
    c.setLayout (new FlowLayout());
    c.add(b1); c.add(b2);
    c.add (text);
  }
        // pentru verificare
 public static void main (String args[ ]) {
   JFrame f = new MFrame();
     f.pack(); f.setVisible(true);
  }
}
```

Numãrul afisat în câmpul text ar putea fi util si altor metode si deci ar putea fi memorat într-o variabilã, desi este oricum memorat ca sir de caractere în câmpul text. Avem o situatie în care trei clase trebuie sã foloseascã în comun o variabilã si sã o modifice. O solutie uzualã este ca variabila externã sã fie transmisã la construirea unui obiect care foloseste aceastã variabilã (ca parametru pentru constructorul clasei). Dacã variabila este de un tip primitiv (*int* în acest caz), constructorul primeste o copie a valorii sale, iar metodele clasei nu pot modifica valoarea originalã. Clasa *Integer* nu este nici ea de folos doarece nu contine metode pentru modificarea valorii întregi continute de un obiect *Integer* si este o clasã finalã, care nu poate fi extinsã cu noi metode. Se poate defini o clasã "Int" ce contine o variabilã de tip *int* si o metodã pentru modificarea sa :

```
class Int {
  int val;
  public Int (int n) { val=n; }                          // constructor
  public int getValue () { return val; }                 // citire valoare
  public void setValue (int n) { val=n; }                // modificare valoare
  public String toString(){return String.valueOf(val); } // pentru afisare
}
```

Clasele pentru obiecte ascultãtor la evenimente sunt aproape la fel:

```
    // receptor la butonul de incrementare
class B1L implements ActionListener {
    Int n;
    JTextField text;
    public B1L (JTextField t, Int n) {
      text=t;  this.n=n;
    }
    public void actionPerformed (ActionEvent ev) {
      int x=n.getValue();
```

```
      n.setValue(++x);  text.setText(n.toString());
    }
  }
```

Clasa cu fereastra principalã a aplicatiei :

```
    // Doua butoane care comanda un camp text
class MFrame extends JFrame  {
   JButton b1 = new JButton ("  +  ");
   JButton b2 = new JButton ("  -  ");
   JTextField text = new JTextField (6);
   Int n= new Int(0);
   public MFrame() {
      Container c = getContentPane();
      b1.addActionListener (new B1L(text,n) );
      b2.addActionListener (new B2L(text,n) );
      c.setLayout (new FlowLayout());
      c.add(b1); c.add(b2);
      text.setText (n.toString()); c.add (text);          // afiseaza val initiala n
   }
}
```

O altã solutie, mai scurtã, porneste de la observatia cã metodele de tratare a evenimentelor diferã foarte putin între ele si cã putem defini o singurã clasã ascultãtor pentru ambele butoane:

```
class BListener implements ActionListener {
   static int n=0;
   JTextField text;
   public BListener (JTextField t) {
      text=t; text.setText (" "+ n);
   }
   public void actionPerformed (ActionEvent ev) {
      String b = ev.getActionCommand();
      if (b.indexOf('+')>=0)   ++n;
      else   --n;
      text.setText(" "+ n);
   }
 }
```

O variantã este definirea clasei cu fereastra aplicatiei ca ascultãtor la evenimente, ceea ce eliminã complet clasele separate cu rol de ascultãtor :

```
class MFrame extends JFrame implements ActionListener {
  JButton b1 = new JButton ("  +  ");
  JButton b2 = new JButton ("  -  ");
  JTextField text = new JTextField (6);
     int n=0;
  public MFrame() {
```

```
    Container c = getContentPane();
    b1.addActionListener (this);
    b2.addActionListener (this);
    c.setLayout (new FlowLayout());
    c.add(b1); c.add(b2);
    text.setText(" "+n); c.add (text);
  }
   public void actionPerformed (ActionEvent ev) {
      Object source =ev.getSource();
      if (source==b1)    ++n;
      else if (source==b2)  --n;
      text.setText(" "+n);
   }
```

Utilizarea de clase incluse

Pentru reducerea numãrului de clase de nivel superior si pentru simplificarea comunicării între clasele ascultător la evenimente putem defini clasele receptor ca niste clase interioare cu nume, incluse în clasa cu fereastra aplicatiei:

```
class MFrame extends JFrame  {
    JButton b1 = new JButton ("  +  ");
    JButton b2 = new JButton ("  -  ");
    JTextField text = new JTextField (6);
    int n= 0;
   public MFrame() {
      Container c = getContentPane();
      b1.addActionListener (new B1L());
      b2.addActionListener (new B2L());
      c.setLayout (new FlowLayout());
      c.add(b1); c.add(b2);
      text.setText(" "+n);
      c.add (text);
   }
         // clase incluse cu nume
   class B1L implements ActionListener {
      public void actionPerformed (ActionEvent ev) {
        text.setText(" "+ ++n);
      }
   }
   class B2L implements ActionListener {
      public void actionPerformed (ActionEvent ev) {
        text.setText(" "+ --n);
      }
  }
```

Definirea de clase incluse anonime reduce si mai mult lungimea programelor, dar ele sunt mai greu de citit si de extins în cazul unor interactiuni mai complexe între

componentele vizuale. O problemã poate fi si limitarea la constructori fãrã argumente pentru clasele anonime. Exemplul anterior rescris cu clase incluse anonime:

```
class MFrame extends JFrame  {
   JButton b1 = new JButton ("  +  "),  b2 = new JButton ("  -  ");
   JTextField text = new JTextField (6);
   int n= 0;
   public MFrame() {
     Container c = getContentPane();
     b1.addActionListener (new ActionListener(){        // definire clasa anonima
       public void actionPerformed (ActionEvent ev) {
         text.setText(" "+ ++n);
       }
     });
     b2.addActionListener (new ActionListener(){        // alta clasa anonima
       public void actionPerformed (ActionEvent ev) {
         text.setText(" "+ --n);
       }
     });
     c.setLayout (new FlowLayout());
     c.add(b1);  c.add(b2);
     text.setText(" "+n);   c.add (text);
   }
}            // clasa MFrame poate contine si metoda "main'
```

Variabilele referintã la componente vizuale “b1”si “b2” sunt definite de obicei în afara functiilor (ca variabile ale clasei), deoarece este probabil ca mai multe metode sã se refere la aceste variabile. Initializarea lor se poate face în afara functiilor (la încãrcarea clasei) deoarece se foloseste un singur obiect fereastrã.

Pe de altã parte, functia “main” sau alte metode statice nu pot folosi direct variabilele referintã la butoane si la alte componente vizuale, dacã aceste variabile nu sunt definite ca variabile statice.

Clase generator si clase receptor de evenimente

Rolul unui buton sau unei alte componente se realizeazã prin metoda “actionPerformed” de tratare a evenimentelor generate de buton, dar care apare într-o altã clasã, diferitã de clasa buton. De aceea, s-a propus definirea de subclase specializate pentru fiecare componentã Swing folosite într-o aplicatie. O astfel de clasã contine si metoda “actionPerformed”.Obiectele acestor clase sunt în acelasi timp sursa evenimentelor dar si ascultãtorul care tratează evenimentele generate. Exemplu:

```
    // Buton "+"
class IncBut extends JButton implements ActionListener {
  JTextField txt;
  public IncBut (JTextField t) {
    super(" + ");  txt=t;
```

```
    addActionListener(this);
  }
  public void actionPerformed (ActionEvent ev) {
    int x=Integer.parseInt(txt.getText());
    txt.setText(String.valueOf(++x));
  }
}
    // Buton "-"
class DecBut extends JButton implements ActionListener {
 . . .
}
    // Doua butoane care comanda un camp text
class MFrame extends JFrame  {
  JButton b1,b2;
  JTextField text = new JTextField (6);
  public MFrame() {
    text.setText("0");
    b1= new IncBut(text);   b2= new DecBut(text);
    Container c = getContentPane();
    c.setLayout (new FlowLayout());
    c.add(b1);  c.add(b2); c.add (text);
  }
    ...  // functia "main"
}
```

Comunicarea între obiectele acestor clase specializate se realizează fie prin referinte transmise la construirea obiectului (sau ulterior, printr-o metodă a clasei), fie prin includerea claselor respective într-o clasă anvelopă.

O variantă a acestei solutii este cea în care de defineste o singură metodă "actionListener" (mai exact, câte o singură metodă pentru fiecare tip de eveniment), care apelează o altă metodă din obiectele buton (sau alt fel de obiecte vizuale specializate). Metoda dispecer de tratare evenimente poate fi inclusă în clasa derivată din *JFrame*. Alegerea metodei "execute" de tratare efectivă a unui tip de eveniment se face prin polimorfism, functie de sursa evenimentului. Exemplu :

```
    // interfata comuna obiectelor vizuale care execută comenzi
interface Command {
  public void execute();
}
    // Buton de incrementare
class IncBut extends JButton implements Command {
   JTextField txt;
   public IncBut (JTextField t) {
     super(" + "); txt=t;
     txt.setText("0");
   }
   public void execute () {
     int x=Integer.parseInt(txt.getText());
     txt.setText(String.valueOf(++x));
```

```
    }
  }
     // Un buton care comanda un camp text
class MFrame extends JFrame implements ActionListener {
   JButton b1;
   JTextField text = new JTextField (6);
   public MFrame() {
      b1= new IncBut (text); b1.addActionListener(this);
      Container c = getContentPane();
      c.setLayout (new FlowLayout());
      c.add(b1); c.add (text);
   }
   public void actionPerformed (ActionEvent ev) {
      Command cmd = (Command) ev.getSource();
      cmd.execute();
   }
}
```

Reducerea cuplajului dintre clase

In examinarea unor variante pentru aplicatia anterioarã am urmãrit reducerea lungimii codului sursã si al numãrului de clase din aplicatie, dar aceastea nu reprezintã indicatori de calitate ai unui program cu obiecte si aratã o proiectare fãrã perspectiva extinderii aplicatiei sau reutilizãrii unor pãrti si în alte aplicatii.

Un dezavantaj comun solutiilor anterioare este cuplarea prea strânsã între obiectele “ascultãtor” la butoane si obiectul câmp text unde se afiseazã rezultatul modificãrii ca urmare a actionãrii unui buton: metoda “actionPerformed” apeleazã direct o metodã dintr-o altã clasã (“setText” din *JTextField*). Desi programarea este mai simplã, totusi tratarea evenimentului de buton este specificã acestei aplicatii iar clasele ascultãtor nu pot fi reutilizate si în alte aplicatii.

Intr-o aplicatie cu multe componente vizuale care interactioneazã este mai dificil de urmãrit si de modificat comunicarea directã dintre obiecte. De aceea s-a propus o schemã de comunicare printr-o clasã intermediar (“mediator”), singura care cunoaste rolul jucat de celelalte clase. Fiecare din obiectele cu rol în aplicatie nu trebuie sã comunice decât cu obiectul mediator, care va transmite comenzile necesare modificãrii unor componente de afisare.

Fiecare dintre obiectele care comunicã prin mediator trebuie sã se înregistreze la mediator, care va retine câte o referintã la fiecare obiect cu care comunicã.

In exemplul cu douã butoane si un câmp text, obiectul mediator este informat de actionarea unuia sau altuia dintre butoane si comandã modificarea numãrului afisat în câmpul text. Pentru reducerea lungimii codului sursã înregistrarea obiectului câmp text la mediator se face în constructorul clasei “MFrame”:

```
     // Buton "+"
class IncBut extends JButton implements ActionListener {
    Mediator med;
    public IncBut (Mediator m) {
```

```
      super(" + ");    med=m;
      addActionListener(this);
    }
    public void actionPerformed (ActionEvent ev) {
      med.inc();
    }
  }
     // Buton "-"
class DecBut extends JButton implements ActionListener {
    Mediator med;
    public DecBut (Mediator m) {
      super(" - ");    med=m;
      addActionListener(this);
    }
    public void actionPerformed (ActionEvent ev) {
        med.dec();
    }
  }
     // clasa mediator
class Mediator {
  private JTextField txt;
  public void registerTxt (JTextField t) {  txt=t;  }
  public void init() {  txt.setText("0");  }
  public void inc() {
    int x=Integer.parseInt(txt.getText());
    txt.setText(String.valueOf(x+1));
  }
  public void dec() {
    int x=Integer.parseInt(txt.getText());
    txt.setText(String.valueOf(x-1));
  }
}
     // Doua butoane care comanda un cimp text
class MFrame extends JFrame  {
    JButton b1,b2;
    JTextField text = new JTextField (6);
    Mediator m = new Mediator();
   public MFrame() {
     b1= new IncBut (m);
     b2= new DecBut (m);
     m.registerTxt (text);
     Container c = getContentPane();
     c.setLayout (new FlowLayout());
     c.add(b1);  c.add(b2); c.add (text);
     m.init();
   }
}
```

Comunicarea dintre obiecte se face aici prin apel direct de metode: obiectele buton apeleazã metodele “inc” si “dec” ale obiectului mediator, iar mediatorul apeleazã

metoda “setText” a obiectului *JTextField* folosit la afisare. Interactiunea dintre un obiect vizual si mediator poate avea loc în ambele sensuri; de exemplu un buton care comandã mediatorului o actiune dar care poate fi dezactivat de mediator.

O variantã de realizare a aplicatiei oarecum asemãnãtoare foloseste schema de clase "observat-observator": butoanele nu actioneazã direct asupra câmpului text, ci sunt obiecte observate. Obiectul observator actioneazã asupra campului text, în functie de obiectul observat (butonul) care l-a notificat de modificarea sa.

Comunicarea prin evenimente si clase “model”

O alternativã la obiectul mediator este un obiect “model”, care diferã de un mediator prin aceea cã modelul genereazã evenimente în loc sã apeleze direct metode ale obiectelor comandate.

O clasã “model” seamãnã cu o componentã vizualã prin aceea cã poate genera evenimente, dar diferã de o componentã Swing prin aceea cã evenimentele nu au o cauzã externã (o actiune a unui operator uman) ci sunt cauzate de mesaje primite de la alte obiecte din program. Inregistrarea unui obiect ascultãtor la un obiect model se face la fel ca si la o componentã vizualã.

Putem folosi o clasã model existentã sau sã ne definim alte clase model cu o comportare asemãnãtoare dar care diferã prin datele continute si prin tipul evenimentelor. In exemplul nostru clasa model trebuie sã memoreze un singur numãr (un obiect) si putem alege între *DefaultListModel* (care contine un vector de obiecte *Object*) sau *DefaultSingleSelectionModel* (care contine un indice întreg).

Programul urmãtor foloseste clasa *DefaultSingleSelectionModel*; numãrul continut poate fi citit cu “getSelectedIndex” si modificat cu metoda “setSelectedIndex”; se genereazã eveniment de tip *ChangeEvent* la modificarea valorii din obiectul model. Pentru scurtarea codului am definit o subclasã a clasei model, cu nume mai scurt:

```
class SModel extends DefaultSingleSelectionModel {
  public void setElement (int x) {
    setSelectedIndex(x);
  }
  public int getElement () {
    return getSelectedIndex();
  }
}
     // Buton "+"
class IncBut extends JButton implements ActionListener {
    SModel mod;
    JTextField txt;
    public IncBut (SModel m, JTextField txt) {
       super(" + ");
       mod=m; this.txt=txt;
       addActionListener(this);
    }
    public void actionPerformed (ActionEvent ev) {
       int n = Integer.parseInt(txt.getText().trim());
```

```
        mod.setElement(n+1);
    }
 }
     // tratare evenimente generate de model
class TF extends JTextField implements ChangeListener {
  public TF (int n) {
    super(n);
  }
  public void stateChanged (ChangeEvent ev) {
    SModel m = (SModel) ev.getSource();
    int x = m.getElement();
    setText (String.valueOf(x));
  }
}
     // Doua butoane care comanda un camp text
class MFrame extends JFrame  {
    JButton b1,b2;
    TF text = new TF(6);
    SModel m = new SModel();
    public MFrame() {
      text.setText("0");
      b1= new IncBut (m,text);
      b2= new DecBut (m,text);
      m.addChangeListener (text);
     Container c = getContentPane();
     c.setLayout (new FlowLayout());
     c.add(b1); c.add(b2); c.add (text);
   }... // main
}
```

Pentru exemplul anterior putem sã ne definim o clasã model proprie, care sã continã o variabilã de tipul general *Object* si sã genereze evenimente de tip *ActionEvent,* prin extinderea clasei *AbstractListModel,* care asigurã partea de generare a evenimentelor si de gestiune a listei de ascultãtori la evenimente.

12. Componente Swing cu model

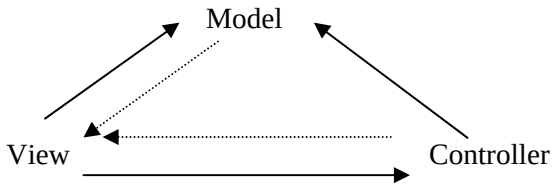
Arhitectura MVC

Arhitectura MVC (Model-View-Controller) separã în cadrul unei componente vizuale cele trei functii esentiale ale unui program sau ale unui fragment de program: intrãri, prelucrãri, iesiri.

Partea numitã “model” reprezintã datele si functionalitatea componentei, deci defineste starea si logica componentei. Imaginea (“view”) redã într-o formã vizualã modelul, iar comanda (“controller”) interpretează gesturile utilizatorului si actionează asupra modelului (defineste “comportarea” componentei). Poate exista si o legãturã directã între partea de control si partea de redare, în afara legãturilor dintre model si celelalte douã pãrti (imagine si comandã).

Comunicarea dintre cele trei pãrti ale modelului MVC se face fie prin evenimente, fie prin apeluri de metode. Modelul semnalizează pãrtii de prezentare orice modificare în starea sa, prin evenimente, iar partea de imagine poate interoga modelul, prin apeluri de metode. Partea de comandã este notificatã prin evenimente de actiunile (“gesturile”) operatorului uman si modificã starea modelului prin apeluri de metode ale obiectului cu rol de “model”; în plus poate apela direct si metode ale obiectului de redare (pentru modificarea imaginii afisate).

In figura urmãtoare liniile continue reprezintã apeluri de metode iar liniile punctate reprezintã evenimente transmise între clase.



De exemplu, un buton simplu este modelat printr-o variabilã booleanã cu douã stãri (apãsat sau ridicat), este comandat printr-un clic de mouse (si, eventual, printr-o tastã) si are pe ecran imaginea unei ferestre dreptunghiulare (rotunde), cu sau fãrã contur, cu text sau cu imagine afisatã pe buton. Imaginea butonului reflectã de obicei starea sa, prin crearea impresiei de buton în pozitie ridicatã sau coborâtã. Este posibil ca imaginea butonului sã fie modificatã direct de actiunea operatorului (de controler) si nu indirect prin intermediul modelului. Legãtura de la imagine la controler constã în aceea cã trebuie mai întâi pozitionat mouse-ul pe suprafata butonului si apoi apãsat butonul de la mouse.

Un obiect vizual folositã într-o interfatã graficã îndeplineste mai multe functii:

- Prezintã pe ecran o imagine specificã (controlabilã prin program).
- Preia actiunile transmise prin mouse sau prin tastaturã obiectului respectiv.
- Modificã valorile unor variabile interne ca rãspuns la intrãri (sau la cereri exprimate prin program) si actualizează corespunzãtor aspectul componentei pe ecran.

Anumite componente Swing se pot folosi în douã moduri:
- Ca obiecte unice asemãnãtoare componentelor AWT (ex. *JButton, JTextField* etc.).
- Ca grupuri de obiecte ce folosesc o variantã a modelului MVC cu numai doi participanti: o clasã model si o clasã "delegat" care reuneste functiile de redare si de control pentru a usura sarcina proiectantului, deoarece comunicarea dintre controler si imagine poate fi destul de complexã.

Componentele *JList, JTable, JTree, JMenuBar* includ întotdeauna un obiect model care poate fi extras printr-o metodã "getModel" pentru a se opera asupra lui.

Un model este o structurã de date care poate genera evenimente la modificarea datelor si contine metode pentru adãugarea si eliminarea de "ascultãtori" la evenimentele generate de model.

Utilizarea unui model de listã

Componenta vizualã *JList* afiseazã pe ecran o listã de valori, sub forma unei coloane, si permite selectarea uneia dintre valorile afisate, fie prin mouse, fie prin tastele cu sãgeti. Datele afisate pot fi de orice tip clasã si sunt memorate într-un obiect colectie (Vector) sau model; o referintã la acest obiect este memoratã în obiectul *JList* de constructorul obiectului *JList*. Existã mai multi constructori, cu parametri de tipuri diferite: vector intrinsec, obiect *Vector* sau obiect *ListModel*. Exemple:

```
String v ={"unu","doi","trei"};
JList list1 = new JList (v);                    // vector intrinsec
Vector vec = new Vector ( Arrays.asList(v));
JList list2 = new JList(vec);                   // obiect de tip Vector
JList list3 = new Jlist (new DefaultListModel());
```

Diferenta între un obiect *JList* cu model si unul fãrã model apare la modificarea datelor din vector sau din model:

Modificarea datelor din model (prin metode ca "addElement", "setElementAt", "removeElementAt") se reflectã automat pe ecran, deoarece obiectul *JList* este ascultãtor la evenimentele generate de model.

Modificarea datelor din vector (vector intrinsec sau obiect *Vector*) nu are efect asupra datelor afisate în obiectul *JList* decât dacã se apeleazã la fiecare modificare si metoda "setListData", care transmite noul vector la obiectul *JList*.

Putem sã nu folosim obiect model atunci când lista este folositã doar pentru selectia unor elemente din listã iar continutul listei afisate nu se mai modificã. Trebuie observat cã orice constructor creeazã un obiect model simplu, dacã nu primeste unul ca argument, iar metoda "getModel" poate fi apelatã indiferent cum a fost creatã lista. Modelul creat implicit are numai metode de acces la date ("getElementAt", "getSize") si nu permite modificarea datelor din acest model (date transmise printr-un vector).

De obicei lista este afisatã într-un panou cu derulare (*JScrollPanel*) pentru a permite aducerea pe ecran a elementelor unei liste oricât de mari (care nu este limitatã la numãrul de linii din panoul de afisare). Ca aspect, o listã *JList* seamãnã cu o zonã

text *JTextArea*, dar permite selectarea unuia sau mai multor elemente din listã.Un obiect *JList* genereazã douã categorii de evenimente:
- Evenimente cauzate de selectarea unui element din listã.
- Elemente cauzate de modificarea continutului listei.

Selectarea unui element dintr-o listã produce un eveniment *ListSelectionEvent*, tratat de o metodã "valueChanged" dintr-un obiect compatibil cu interfata *ListSelectionListener*. Printre alte informatii furnizate de modelul de listã este si indicele elementului selectat (metoda "getSelectedIndex"), ceea ce permite clasei "handler" accesul la elementul selectat.

Exemplul urmãtor aratã cum se poate programa cel mai simplu o listã de selectie; elementul selectat este afisat într-un câmp text, pentru verificarea selectiei corecte.

```
class ListSel extends JFrame {
  JTextField field;                                    // ptr. afisare element selectat
  JList list;                                          // lista de selectie
     // clasa adaptor intre JList si JTextField (clasa inclusa)
  class LSAdapter implements ListSelectionListener {
       int index=0;
     public void valueChanged(ListSelectionEvent e) {
       ListModel model = list.getModel();
        index = list.getSelectedIndex();                    // indice element selectat
        String sel = (String)model.getElementAt(index);        // element selectat
       field.setText(sel);                                 // afisare elem. selectat
     }
    }
  public ListSel() {
    String[ ] sv ={"unu","doi","trei","patru"};
    list = new JList(sv);                                  // creare lista de selectie
    list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
    list.setSelectedIndex(0);
    list.addListSelectionListener(new LSAdapter());        // receptor evenim selectie
    Container cp = getContentPane();
    cp.setLayout (new FlowLayout());
    JScrollPane pane = new JScrollPane(list);              // panou cu defilare ptr lista
    cp.add(pane);
    field =new JTextField(10); field.setEditable(false);   // creare cimp text
    cp.add(field);
  }
  public static void main(String s[ ]) {
       ListSel ls = new ListSel();
       ls.pack(); ls.setVisible(true);                     // afisare lista si cimp text
  }
}
```

Existã si alte posibilitãti de grupare în clase a functiilor necesare acestei aplicatii: clasa fereastrã principalã (derivatã din *JFrame*) separatã de clasa pentru tratarea evenimentului de selectie din listã si separatã de clasa cu functia "main".

Programul anterior nu permite modificarea continutului listei de selectie deoarece interfata *ListModel* nu prevede metode de adãugare sau de eliminare obiecte din model. Solutia acestei probleme se aflã în crearea unui obiect model explicit si transmiterea lui ca argument la construirea unui obiect *JList*. Cel mai simplu este sã definim modelul nostru de listã ca obiect al unei clase derivate din *DefaultListModel* (care include un vector), dar putem sã definim si o clasã derivatã din *AbstractListModel* sau care implementeazã interfata *ListModel* si care contine un alt tip de colectie (de exemplu, o listã înlãntuitã de elemente si nu un vector).

Exemplul urmãtor aratã cum se pot adãuga elemente la sfârsitul unei liste de selectie, cu un buton pentru comanda de adãugare (“Add”) si un câmp text unde se introduce elementul ce va fi adãugat la listã.

```
class ListAdd extends JFrame {
  private JList list;
  private DefaultListModel listModel;
  private JButton addBut=new JButton("Add");
  private JTextField toAdd = new JTextField(10);
         // constructor
  public ListAdd() {
    listModel = new DefaultListModel();                    // model de date pentru lista
    list = new JList(listModel);
    JScrollPane lpane = new JScrollPane(list);          // lista in panou cu derulare
    addBut.addActionListener(new AddListener());    // receptor la buton
    toAdd.addActionListener(new AddListener());      // receptor la camp text
    Container cpane = getContentPane();
    cpane.setLayout( new FlowLayout());
    cpane.add(toAdd);                              // adauga camp text
    cpane.add(addBut);                             // adauga buton
    cpane.add(lpane);                              // adauga panou cu derulare pentru lista
 }
     // receptor la buton si la campul text (clasã inclusã)
 class AddListener implements ActionListener {
   public void actionPerformed(ActionEvent e) {
      if (toAdd.getText().equals("")) {               // verifica ce este in campul text
         Toolkit.getDefaultToolkit().beep(); return;        // daca nimic in campul text
      }
      listModel.addElement(toAdd.getText());   // adauga element la model lista
      toAdd.setText ("");                              // sterge camp text
   }
 }
}
```

Exemplul anterior poate fi extins cu un buton de stergere element selectat din listã si o clasã care implementeazã interfata *ListSelectionListener*, cu o metodã “valueChanged” ; un obiect al acestei clase se va înregistra ca ascultãtor la listã.

Modificarea vectorului din modelul de listã genereazã evenimente pentru obiecte înregistrate ca ascultãtori la listã: metoda “addElement” din clasa *DefaultListModel*

apeleazã metoda “fireIntervalAdded”, metoda “removeElement” apeleazã metoda “fireIntervalRemoved” si “setElementAt” apeleazã metoda “fireContentsChanged”.

Un observator la modificări ale listei trebuie sã implementeazã interfata *ListDataListener*, deci sã defineascã metodele “intervalAdded”, “intervalRemoved” si “contentsChanged”, toate cu argument de tip *ListDataEvent*, care contine informatii despre sursa, tipul evenimentului si pozitia din vector unde s-a produs modificarea. Putem defini o clasã adaptor cu implementări nule pentru toate cele trei metode :

```
class ListDataAdapter implements ListDataListener {
  public void intervalRemoved (ListDataEvent ev) { }
  public void intervalAdded (ListDataEvent ev) { }
  public void contentsChanged (ListDataEvent ev) { }
}
```

Exemplul urmãtor aratã cum putem reactiona la stergerea unui element din listã prin afisarea dimensiunii listei dupã stergere:

```
class LFrame extends JFrame {
  String[] a ={" 1"," 2"," 3"," 4"," 5"};
  DefaultListModel model = new DefaultListModel();   // Model
  JTextField result = new JTextField(4);     // rezultat modificare
  JButton cb = new JButton("Delete");        // comanda operatia de stergere
  JList list;
  static int index;                          // pozitia elementului sters
         // ascultator la butonul de stergere
  class BAL implements ActionListener {
    public void actionPerformed (ActionEvent ev) {
      if (model.size() >0)
        model.removeElementAt(index);
    }
  }
         // ascultator la selectie din lista
  class LSL implements ListSelectionListener {
       public void valueChanged(ListSelectionEvent e) {
          index = list.getSelectedIndex();
       }
   }
         // ascultator la stergere din lista
  class LDL extends ListDataAdapter {
    public void intervalRemoved (ListDataEvent ev) {
         result.setText(" "+ model.size());
    }
   }
     // constructor
  public LFrame () {
    for (int i=0;i<a.length;i++)
      model.addElement(a[i]);
    list = new JList(model);
    list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
    list.addListSelectionListener(new LSL());
```

```
    Container c = getContentPane();
    c.setLayout (new FlowLayout ());
    c.add(new JScrollPane(list));
    c.add(cb); c.add(result);
    cb.addActionListener (new BAL());
    model.addListDataListener( new LDL());
    result.setText(model.size()+" ");
  }
```

Familii deschise de clase în JFC

Componentele vizuale cu text din JFC constituie un bun exemplu de proiectare a unor clase care satisfac două cerinte aparent contradictorii: posibilitatea de modificare de către programatori (pentru extinderea functionalitãtii sau pentru performante mai bune) si simplitatea de utilizare a unei versiuni "standard", uzuale, a clasei respective.

O componentã vizualã cu text se obtine simplu prin instantierea unei clase direct utilizabile (*JList, JTable* s.a.). Aceste clase reunesc două pãrti distincte : modelul folosit în accesul la date si partea de afisare pe ecran a datelor din model. Programatorul poate interveni în alegerea colectiei de date folosite de model si chiar în stabilirea functiilor realizate de model (pe lângã functiile minimale, standard).

Interfete si clase folosite în proiectarea componentelor vizuale cu model:

1) O interfatã ce stabileste cerintele minimale pentru modelul logic al datelor : *ListModel, TableModel, TreeModel*. Exemplu:

```
public interface ListModel {
  int getSize();                                    // lungime lista
  Object getElementAt(int index);                   // valoare din pozitia index
  void addListDataListener(ListDataListener lsn);
  void removeListDataListener(ListDataListener lsn);
}
```

2) O clasã abstractã care implementeazã partea de gestiune a listei de ascultatori si realizeazã declansarea evenimentelor specifice acestui model:

```
public abstract class AbstractListModel implements ListModel {
  protected EventListenerList listenerList = new EventListenerList();
  public void addListDataListener(ListDataListener l) {
      listenerList.add(ListDataListener.class, l);
  }
  public void removeListDataListener(ListDataListener l) {
      listenerList.remove(ListDataListener.class, l);
  }
        // la modificare continut listã
  protected void fireContentsChanged(Object source, int from, int to) {
    Object[] listeners = listenerList.getListenerList();
    ListDataEvent e = null;
    for (int i = listeners.length - 2; i >= 0; i -= 2) {
       if (e == null)
```

```
            e=new ListDataEvent(source,ListDataEvent.CONTENTS_CHANGED,from, to);
            ((ListDataListener)listeners[i+1]).contentsChanged(e);
        }
    }
          // la adaugarea unor elemente la lista
    protected void fireIntervalAdded(Object source, int from, int to) {
       Object[] listeners = listenerList.getListenerList();
       ListDataEvent e = null;
       for (int i = listeners.length - 2; i >= 0; i -= 2) {
           if (e == null)
            e = new ListDataEvent(source, ListDataEvent.INTERVAL_ADDED, from, to);
            ((ListDataListener)listeners[i+1]).intervalAdded(e);
        }
    }
          // la eliminarea unor elemente din lista
    protected void fireIntervalRemoved(Object source, int from, int to) {
        Object[] listeners = listenerList.getListenerList();
        ListDataEvent e = null;
        for (int i = listeners.length - 2; i >= 0; i -= 2) {
          if (e == null)
            e= new ListDataEvent(source,ListDataEvent.INTERVAL_REMOVED,from, to);
          ((ListDataListener)listeners[i+1]).intervalRemoved(e);
        }
     }
}
```

3) O clasã instantiabilã care realizeazã o implementare standard a colectiei de date folosite de model. De exemplu, clasa *DefaultListModel* foloseste un obiect de tip *Vector* pentru colectia liniarã de obiecte si preia o parte din metodele clasei *Vector*. Clasa model “deleagã” clasei vector realizarea operatiilor de acces si de modificare a datelor din model.

```
public class DefaultListModel extends AbstractListModel {
   private Vector delegate = new Vector();
   public int getSize() { return delegate.size(); }
   public Object getElementAt(int index) { return delegate.elementAt(index); }
      // metode ptr modificare date din model
   public void addElement(Object obj);
   public boolean removeElement (Object obj);
   ...
}
```

4) O clasã pentru obiecte vizuale, care generează evenimente cauzate de gesturi ale operatorului uman. Clasa *JList* contine o variabilã de tip *ListModel* initializatã de fiecare constructor :

```
public class JList extends JComponent implements Scrollable, Accessible {
   private ListModel dataModel;          // model folosit de clasa JList
   . . .                                 // alte variabile ale clasei JList
```

```
   public JList(ListModel dataModel) {  // constructor cu parametru ListModel
      this.dataModel = dataModel;  . . .          // inlocuire variabila interfata
   }
   public JList(final Object[] listData) {       // constructor cu param. vector
      this ( new AbstractListModel() {             // apeleaza alt constructor
               public int getSize() { return listData.length; }
               public Object getElementAt(int i) { return listData[i]; }
              }
      );
   }
   public JList() {
     this ( new AbstractListModel() {
          public int getSize() { return 0; }
          public Object getElementAt(int i) { return "No Data Model"; }
       }
     );
   }
. . .     // alte metode din clasa JList
}
```

Variabila interfatã “dataModel” poate fi modificatã si prin metoda “setModel” si poate fi cititã prin metoda “getModel”. Exemplu de metodã din clasa *JList* :

```
public Object getSelectedValue() {           // extrage din listã obiectul selectat
  int i = getMinSelectionIndex();             // indice element selectat
  return (i == -1) ? null : getModel().getElementAt(i);      // valoare element
}
```

O clasã model de listã (si componenta *JList*, care foloseste acest model) genereazã trei tipuri de evenimente. Interfata *ListDataListener*, la care trebuie sã adere obiectele ascultãtor interesate de modificãrile efectuate asupra listei, contine trei metode: contentsChanged, intervalAdded si intervalRemoved. In plus, o listã *JList* poate avea si un obiect ascultãtor la selectarea unui element din listã (*ListSelectionEvent*), obiect compatibil cu interfata *ListSelectionListener*.
Metodele de tip “fireXXX” apeleazã metodele de tratare a evenimentelor din obiectele ascultãtor înregistrate.

Lantul de apeluri care produce activarea unei metode scrise de programator ca urmare a unei modificãri a colectiei de date din model este, în cazul listei, urmãtorul:

```
DefaultListModel.addElement -> AbstractModel.fireIntervalAdded ->
        -> (ListDataEvent)->ListDataListener.intervalAdded
```

Metoda “JList.fireSelection” apeleaza metoda “valueChanged” din obiectele ascultator (de tip *ListSelectionListener*), cãrora le transmite un eveniment *ListSelectionEvent*.

La construirea unui obiect *JList* exista urmãtoarele variante pentru programator:
- Sã nu foloseascã un model de listã explicit (apel constructor fãrã nici un parametru); o solutie simplã pentru liste nemodificabile.

- Sã foloseascã un model creat automat ca subclasã a clasei *AbstractListModel* , prin apelul unui constructor cu parametru vector intrinsec sau obiect *Vector*.
- Sã foloseascã un obiect model de tip *DefaultListModel* transmis ca parametru unui constructor al clasei *JList* cu parametru de tip *ListModel*.
- Sã foloseascã un obiect model de un tip definit de utilizator (ca subtip al clasei abstracte) si transmis ca parametru de tip *ListModel* unui constructor al clasei *JList*.

Utilizatorul clasei *JList* are urmãtoarele posibilitãti de a interveni :

- Sã foloseascã un alt model, compatibil cu interfata *ListModel* si care sã continã un alt fel de colectie în loc de vector (de exemplu o listã înlãntuitã)
- Sã extragã obiectul model si sã apeleze metode specifice modelului folosit, dar care nu sunt prezente si în interfata "ListModel" (pentru modificare date din model, de ex.)
- Sã modifice obiectul de redare a elementelor listei pe ecran ("setCellRenderer").

Utilizarea unui model de tabel

Componenta vizualã *JTabl*e afiseazã pe ecran un tabel de celule structurat în linii si coloane, permitând selectarea si editarea de celule, modificarea aspectului tabelului si obtinerea de informatii despre celulele selectate (cu dispozitivul "mouse"). Fiecare coloanã poate avea un titlu. Datele din celule pot fi de tipuri diferite, dar numai tipuri clasã. Dimensiunile coloanelor sunt implicit egale si calculate în functie de numãrul coloanelor, dar pot fi modificate fie prin program (metoda "setPreferredWidth"), fie prin deplasarea ("tragerea"="dragging") marginilor coloanelor folosind un mouse.

Construirea unui obiect *JTable* se poate face folosind un vector de titluri si o matrice de celule (de obiecte *Object*), sau folosind un obiect model *TableModel* :

```
JTable (Object[][] date, Object[] numecol);      // cu vectori intrinseci
JTable (Vector date, Vector numecol);              // cu obiecte Vector
JTable (TableModel model);                         // cu obiect model
```

Secventa urmãtoare ilustreazã folosirea acestor constructori:

```
    // tabelul primeste direct datele
String [ ] titles = { "Românã", "Englezã","Germanã"};      // nume de coloane
Object[ ][ ] cells = { {"Unu","One","Eins"}, {"Doi","Two","Zwei"},... };
JTable table1 = new JTable (cells,titles);
    // tabelul primeste un model al datelor
DefaultTableModel model = new DefaultTableModel (cells,titles);
JTable table2 = new JTabel (model);
```

Un obiect *JTable* include un obiect model, care poate fi extras si manipulat prin metode suportate de modelul tabelar. Solutia unui tabel fãrã un obiect model explicit este mai simplã dar mai putin flexibilã: toate celulele sunt editabile (modificabile), toate tipurile de date sunt prezentate la fel, nu sunt posibile alte surse de date decât vectori.

Se poate defini o clasã model care sã implementeze interfata *TableModel*, sau sã extindã una din clasele model existente *AbstractTableModel* sau *DefaultTableModel*. De exemplu, pentru a interzice editarea (modificarea) continutului tabelului putem sã redefinim o metodã din clasa *DefaultTableModel* sau sã definim mai multe metode din clasa *AbstractTableModel*:

```
        // tabel nemodificabil prin extindere DefaultTableModel
class MyTableModel extends DefaultTableModel {
  public TableModel2 (Object[ ][ ] cells, Object[ ] cnames) {
      super (cells,cnames);
  }
  public boolean isCellEditable (int row,int col) {
      return false;              // nici o celula editabila
  }
}
        // tabel nemodificabil prin extindere AbstractTableModel
class MyTableModel extends AbstractTableModel {
  Object [ ] titles;              // titluri coloane
  Object [ ][ ] cells ;                 // continut celule
     public MyTableModel ( Object c[][], Object t[]) {
       cells=c; titles=t;
     }
     public int getColumnCount() { return titles.length; }        // numar de coloane
     public int getRowCount() { return cells.length; }            // numar de linii
     public Object getValueAt (int row,int col) { return cells[row][col]; }
     public String getColumnName(int col) { return titles[col].toString();}
     public void setValueAt (Object val,int row,int col) {
        cells[row][col] = val.toString();
     }
}
```

Pentru ambele variante crearea si afisarea tabelului este identicã :

```
class MyTable extends JFrame {
   . . .   // vectori de date
   JTable table = new JTable(new MyTableModel(cells,titles));
   JScrollPane scrollPane = new JScrollPane(table);
   getContentPane().add(scrollPane, BorderLayout.CENTER);
   JFrame f = new MyTable();
     . . .
}
```

Modelul unui tabel poate fi extras (metoda “getModel”) si manipulat pentru citirea datelor din fiecare celulã (metoda “getValueAt(row,col)”) sau modificarea acestor date.

Pentru operatii cu o coloanã dintr-un tabel este prevãzut modelul unei coloane: interfata *ColumnModel* si modelul implicit *DefaultColumnModel*.

Un tabel poate genera mai multe tipuri de evenimente :

- Evenimente produse la selectarea unei linii, unei coloane sau unei celule.
- Evenimente produse de modificarea datelor din tabel
- Evenimente produse de modificarea structurii tabelului

In mod implicit este permisã numai selectarea de linii dintr-un tabel, dar existã metode pentru activarea selectiei de coloane sau de celule individuale. In exemplul urmãtor se permite selectarea de celule si se afiseazã în douã etichete *JLabel* numãrul liniei si al coloanei selectate.

```
class TableSel extends JFrame {
  public TableSel() {
    Object[][] data = { ...};  String[] columnNames = { ...};
    final JLabel selrow = new JLabel("    ");           // linie selectata
    selrow.setBorder (new EtchedBorder());
    final JLabel selcol = new JLabel("    ");           // coloana selectata
    selcol.setBorder (new EtchedBorder());
    final JTable table = new JTable(data, columnNames);
    table.setPreferredScrollableViewportSize(new Dimension(500, 70));
    table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
    table.setColumnSelectionAllowed(true);
    table.setCellSelectionEnabled(true);
    ListSelectionModel rowSM = table.getSelectionModel();
    rowSM.addListSelectionListener(new ListSelectionListener() {
      public void valueChanged(ListSelectionEvent e) {
        ListSelectionModel lsm = (ListSelectionModel)e.getSource();
            int row = lsm.getMinSelectionIndex();
        selrow.setText(" "+ row);
      }
    });
    ListSelectionModel colSM = table.getColumnModel().getSelectionModel();
    colSM.addListSelectionListener(new ListSelectionListener() {
      public void valueChanged(ListSelectionEvent e) {
        ListSelectionModel lsm = (ListSelectionModel)e.getSource();
              int col = lsm.getMinSelectionIndex();
        selcol.setText(" " + col);
      }
    });
    JScrollPane scrollPane = new JScrollPane(table);
       Container pane= getContentPane();
    pane.setLayout(new FlowLayout());
    pane.add(scrollPane);
    pane.add(selrow); pane.add(selcol);
  }
  public static void main(String[] args) {
    TableSel frame = new TableSel();
    frame.pack(); frame.setVisible(true);
  }
}
```

Utilizarea unui model de arbore

Clasa *JTree* permite afisarea unor structuri arborescente (ierarhice), cu posibilitatea de a extinde orice nod prin subarborele sãu, de a comprima un subarbore la rãdãcina sa si de a selecta orice nod din arbore prin “mouse”. O aplicatie va trata evenimentele de selectare a unor noduri sau de modificare structurã arbore sau de extindere -contractie noduri corespunzãtor scopului aplicatiei.

Clasa *JTree* foloseste un model de arbore, adicã o structurã cu datele ce vor fi afisate si care poate genera evenimente. Modelul de arbore poate fi creat de utilizator înainte de a construi obiectul *JTree* sau este creat în interiorul clasei *JTree* pe baza unei alte colectii de date furnizate de utilizator (vector intrinsec, obiect Vector, etc.).

Modelul de arbore este fie un obiect din clasa *DefaultTreeModel*, fie obiect al unei clase definite de utilizator, dar care implementeazã interfata *TreeModel*. Acest model genereazã numai evenimente legate de modificarea datelor; pentru a trata evenimente legate de selectia unor noduri trebuie folosit un model *TreeSelectionModel*.

Clasa model de arbore foloseste un obiect model de nod, ca rãdãcinã a arborelui. Ca model de nod se poate folosi un obiect din clasa *DefaultMutableTreeNode* sau obiecte din clase definite de utilizatori care implementeazã interfata *TreeNode* (noduri nemodificabile) sau interfata *MutableTreeNode* (cu operatii de modificare). Fiecare nod contine un obiect al aplicatiei (de tip *Object*), legãturi cãtre pãrinte si cãtre fii sãi.

Exemple de metode din interfata *TreeNode*:

```
int getChildCount();                  // numar de succesori ai acestui nod
TreeNode getChildAt(int childIndex);  // succesor cu indice cunoscut
int getIndex(TreeNode node);          // indicele unui succesor dat
TreeNode getParent();         // nodul parinte al acestui nod
boolean isLeaf();                     // daca acest nod este o frunza
Enumeration children();               // enumerator pentru succesorii acestui nod
```

Exemple de metode din interfata *MutableTreeNode*:

```
void remove(int index);                    // elimina succesor cu indice dat
void remove(MutableTreeNode node); // elimina succesor dat ca nod
void setUserObject(Object object);       // modifica datele din acest nod
void removeFromParent();                   // eliminã acest nod
void setParent(MutableTreeNode newParent);    // modifica parintele acestui nod
```

Clasa *DefaultMutableTreeNode* este o implementare a interfetei *MutableTreeNode* si foloseste câte un vector de succesori la fiecare nod (obiect de tip *Vector*), plus o legãturã cãtre nodul pãrinte. Exemple de metode în plus fatã de iterfete:

```
public Object getUserObject ();               // obtine date din acest nod (orice obiect)
public void add(MutableTreeNode newChild) ;   // adauga acestui nod un fiu
public Enumeration preorderEnumeration() ;    // enumerare noduri din subarbore
public Enumeration postorderEnumeration() ;   // enumerare noduri din subarbore
public Enumeration breadthFirstEnumeration() ;  // enumerare noduri din subarbore
```

Exemple de utilizare a unor obiecte de tip *DefaultMutableTreeNode:*

```
  // clasa ptr un nod de arbore ( cu un nume mai scurt)
class TNode extends DefaultMutableTreeNode {
  public TNode() { super();}
  public TNode (Object info) { super(info);}
}
      // cauta in arborele cu radacina root nodul ce contine un obiect dat
  public static TNode find (TNode root, Object obj) {
    Enumeration e = root.preorderEnumeration();
    while ( e.hasMoreElements()){
       TNode n = (TNode) e.nextElement();
       if (n.getUserObject().equals(obj))
           return n;
    }
    return null;
  }
```

Clasa predefinitã “model de nod” se poate folosi si ca o colectie arborescentã în aplicatii fãrã interfatã graficã, independent de clasa *JTree*. Exemplu de clasã pentru obiecte arbori multicãi cu numele fisierelor dintr-un director dat:

```
     // subclasa ptr arbori multicai de fisiere
class FTree extends DefaultMutableTreeNode {
  public FTree() { super();}
  public FTree (String info) { super(info);}
  public FTree (File dir) {        // arbore cu fisierele din directorul “dir”
    this (dir.getName());          // radacina arbore, cu nume fisier director
    growtree (this, dir);       // adaugare recursiva noduri la acest arbore
  }
     // creare arbore cu fisierele dintr-un director d si din subdirectoarele sale
  static void growtree ( FTree r, File d) {
    if ( ! d.isDirectory( ) ) return;   // iesire daca d nu e fisier director
    String [ ] files=d.list();                  // nume fisiere din directorul d
    for(int i=0;i<files.length;i++){                    // ptr fiecare fisier
       File f = new File(d+"\\"+files[i]);          // creare obiect File
       FTree kid = new FTree (f.getName());  // creare nod ptr nume fisier
       r.add ( kid); kid.setParent(r);                  // leaga nod la arbore
       growtree (kid, f);                     // adauga continut subdirector f
    }
  }
}
```

Continutul acestui arbore poate fi afisat în mod text sau în mod grafic, folosind un obiect al clasei “FTree”. Exemplu de afisare în mod text, folosind metode ale clasei *DefaultMutableTreeNode*, dar fãrã indentare (o singurã coloanã cu numele fisierelor):

```
    public static void main (String arg[ ])  {
       FTree r= new FTree ( new File (arg[0]) );          // arg[0]=nume director radacina
```

```
      Enumeration e = r.preorderEnumeration();       // metoda mostenita
      while (e.hasMoreElements())
        System.out.println( e.nextElement());
    }
```

Pentru o afisare cu indentare, care sã reflecte structura arborescentã a unui sistem de fisiere putem scrie o functie de parcurgere a arborelui. Exemplu:

```
 public void print () {                          // metoda a clasei FTree
   printTree ( (FTree) getRoot(), "");
 }
 static void printTree (FTree r, String ind) { // ind = indentare la fiecare apel
   final String inc ="  ";  // cu cat creste indentarea la un nou nivel
   System.out.println (ind+ r.toString());     // afisare nod radacina
   Enumeration e = r.children();               // fiii nodului r
   ind=ind+inc;                                // creste indentarea la un nou nivel
   while ( e.hasMoreElements())                // afisare recursiva fii
     printTree ((FTree) (e.nextElement()),ind);
 }
```

Pentru afisare în mod grafic, o functie “main” minimalã poate arãta astfel:

```
class FileJTree extends JFrame {
  public static void main (String arg[])  {
     JTree t = new JTree ( new FTree ( new File (arg[0])));
     JScrollPane pane = new JScrollPane (t);
     FileJTree ft = new FileJTree();
     ft.getContentPane().add (pane);
     ft.pack(); ft.show();
  }
}
```

De observat cã arborele afisat de programul anterior nu genereazã evenimente, pentru cã nu foloseste o clasã model *TreeModel* sau *TreeSelectionModel*, dar se pot expanda si contracta noduri prin selectie si dublu clic pe mouse.

13. Java si XML

Fisiere XML în aplicatii cu obiecte

XML este o modalitate standardizatã de a reprezenta date într-un mod independent de sistem (de o platformã) si a cunoscut o largã utilizare în ultimii ani.

Limbajul XML (“Extensible Markup Language”) este un limbaj cu marcaje (“tag” va fi tradus aici prin “marcaj”) si extensibil, în sensul cã multimea de marcaje folosite poate fi definitã în functie de tipul aplicatiei si nu este fixatã dinainte (ca în HTML). Marcajele au rolul de a descrie datele încadrate de marcaje, deci semnificatia sau modul de interpretare a acestor date (si nu modul de prezentare, ca în HTML).

Un fisier XML este un fisier text care contine marcaje; un marcaj este un sir între paranteze unghiulare (‘<’ si ‘>’).

Un prim exemplu este un mic fragment dintr-un fisier XML listã de preturi:

```
<priceList>
  <computer>
    <name> CDC </name>
    <price> 540 </price>
  </ computer >
  <computer>
     <name> SDS </name>
     <price> 495 </price>
  </ computer >
</priceList>
```

Un element este fragmentul cuprins între un marcaj de început si un marcaj de sfârsit, dar pot exista si elemente definite printr-un singur marcaj si cu atribute. Un document XML are o structurã ierahicã, arborescentã, în care un element poate contine alte elemente, care la rândul lor pot contine alte elemente s.a.m.d. Elementele care nu mai include alte elemente contin între marcaje date, dar si elementele complexe pot contine date sub formã de atribute. Exemplu de element cu un atribut:

```
<computer type =”desktop”> . . .     </ computer >
```

Multimea marcajelor folosite într-un document XML este definitã într-un fisier “schemã” XML, folosit la validarea fisierelor care folosesc aceste marcaje. Se folosesc douã tipuri de fisiere schemã: fisiere DTD (Document Type Definition) si fisiere XSD (XML Schema Definition). Exemplu de fisier DTD pentru lista de preturi

```
<!ELEMENT priceList (computer)+>
<!ELEMENT computer (name, price) >
<!ELEMENT name (#PCDATA) >
<!ELEMENT price (#PCDATA) >
```

Fisierul XSD este un fisier XML care foloseste marcaje predefinite . Exemplu:

```
<schema>
 <element name="pricelist">
  <complexType>
   <element ref="computer" minOccurs='1' maxOccurs='unbounded'/>
  </complexType>
 </element>

 <element name="computer">
  <complexType>
   <element ref="name"/>
   <element ref="price"/>
  </complexType>
 </element>

 <element name="name" type='string'/>
 <element name="price" type='string'/>
</schema>
```

Pentru a permite utilizarea de marcaje cu acelasi nume în scheme diferite s-a introdus si în XML conceptul “spatiu de nume”: un spatiu de nume contine marcaje distincte si este declarat la începutul unui fisier schemã. Dacã într-un acelasi fisier XML se folosesc marcaje din mai multe spatii de nume, atunci numele de marcaje trebuie prefixate de un simbol asociat spatiului de nume.

Mentionãm câteva dintre utilizãrile documentelor XML în aplicatii OO : pentru serializare continut obiecte si apeluri de proceduri la distantã, pentru mesaje transmise între calculatoare (mesaje SOAP), pentru fisiere de configurare, fisiere descriptor de componente Java (“deployment descriptor”), fisiere “build” folosite de programul “ant” pentru construire si instalare de aplicatii, s.a.

Multe navigatoare Web si medii integrate de dezvoltare Java (sau pentru alte limbaje OO) includ editoare si parsere XML, permitând si validarea de fisiere XML.

Un parser este un analizor de documente XML care poate crea un obiect cu structura si continutul fisierului XML (parser DOM), sau poate apela anumite functii la detectarea de elemente semnificative în fisierul XML (parser SAX).

XML si orientarea pe obiecte

Un document XML este o instantiere a unei scheme, asa cum un obiect este o instantiere a unei clase. Se pot defini clase corespunzãtoare elementelor XML, iar trecerea între fisiere XML si clase Java (în ambele sensuri) este asiguratã de clasele din interfata API numitã JAXB (Java XML Binding). Clasele Java sunt generate pe baza unei scheme XML (de obicei un fisier DTD).

Aparitia si evolutia limbajului XML a fost contemporanã cu orientarea pe obiecte în programare, ceea ce se vede si în aceea cã singurul model de prelucrare a fisierelor XML standardizat (W3C) este modelul DOM (“Document Object Model”), model independent de limbajul de programare folosit.

In esentã, modelul DOM înseamnã cã rezultatul prelucrãrii unui fisier XML este un arbore ce reflectã structura documentului, cu date în anumite noduri (terminale); standardizarea se referã la existenta unei interfete API unice pentru prelucrarea acestui arbore de cãtre aplicatii, indiferent de limbajul folosit.

Prelucrarea fisierelor XML a constituit o provocare pentru programarea orientatã pe obiecte si, în special pentru limbajul Java în care existau deja predefinite multe clase si tehnici de programare potrivite acestui scop.

Un parser XML este un procesor de documente XML care furnizeazã unei aplicatii continutul fisierului, într-o formã sau alta. Un parser verificã dacã fisierul XML este bine format ("well formed") si poate valida, optional, documentul XML.

Un fisier XML bine format are toate perechile de marcaje incluse corect, la fel cum incluse perechile de paranteze dintr-o expresie sau perechile de acolade din Java. Exemple de erori de utilizare a marcajelor semnalate de un parser:

<a> … … …   (corect este <a>……..)
<a> … … … </c>

Validarea XML necesitã specificarea unui fisier schemã (DTD sau XSD) la începutul documentului XML si înseamnã certificarea faptului cã au fost folosite numai marcajele din schemã, cu relatiile, atributele si tipurile de date descrise în schemã. Validarea mãreste timpul de prelucrare si este inutilã atunci când fisierele XML sunt generate de cãtre programe si nu sunt editate manual.

Un parser bazat pe modelul DOM are ca principal dezavantaj memoria necesarã pentru arborele corespunzãtor unui document XML mai mare (mai ales cã în acest arbore apar noduri si pentru spatiile albe folosite la indentare si la fiecare linie nouã). Spatiile albe nu sunt esentiale pentru întelegerea unui fisier XML si de aceea nici nu apar în fisiere destinate a fi prelucrate numai prin programe si transmise prin retea.

De exemplu, fisierul listã de preturi poate apare si sub forma urmãtoare:

```
<!DOCTYPE priceList SYSTEM "priceList.DTD">
<priceList> <computer><name>CDC</name><price>540</price></computer>
<computer><name>SDS</name><price>495</price> </ computer ></priceList>
```

O altã criticã adusã modelului DOM este aceea cã interfata API nu este total în spiritul orientãrii pe obiecte si nu foloseste clasele colectie si iterator din Java. De aceea s-au propus si alte variante de modele arbore : JDOM si DOM4J (J de la Java).

Avantajul unui parser care creeazã un obiect arbore este acela cã permite operatii de cãutare, de modificare în arbore si de creare a unui alt document XML din arbore.

O abordare total diferitã fatã de modelul DOM a dat nastere programelor parser de tip SAX ("Simple API for XML"), care folosesc functii "callback" apelate de parser dar scrise de utilizator, ca parte dintr-o aplicatie. Pe mãsurã ce parserul analizeazã fisierul XML, el genereazã evenimente si apeleazã metode ce respectã anumite tipare. Exemple de evenimente SAX: întâlnire marcaj de început, întâlnire marcaj de sfârsit, întâlnirea unui sir de caractere între marcaje, etc. La apelul metodelor de tratare a evenimentelor se transmit si alte informatii, cum ar fi continutul marcajului si atribute

Un parser SAX este mai rapid, consumã mai putinã memorie si este mai usor de folosit (nu necesitã navigarea printr-un arbore), dar nu permite modificarea de fisiere XML si nici operatii repetate de cãutare în fisier într-un mod eficient. Este util atunci când o aplicatie este interesatã numai de anumite sectiuni dintr-un document XML si nu de întreg continutul. O aplicatie poate construi un arbore pe baza informatiilor primite de la un parser SAX.

Pentru cã nu existã un singur parser SAX sau DOM pentru Java si pentru a putea utiliza orice parser cu minim de modificări, se practicã obtinerea unui obiect parser prin apelul unei metode fabricã si nu prin instantierea directã a unei singure clase.

Un alt standard, materializat într-un API Java, este XSLT (XML Stylesheet Language for Transformation), care permite specificarea unor transformări asupra unui document XML în vederea conversiei sale într-un alt format (de ex. HTML). Un obiect convertor ("transformer") primeste la creare un obiect sursã si un obiect rezultat, iar metoda "transform" face transformările necesare de la sursã la rezultat.

Interfetele "Source" si "Result" sunt implementate de clase "DOMSource" si "DOMResult", "SAXSource" si "SaxResult" sau "StreamSource", "StreamResult" (fluxuri de intrare-iesire ca sursã si rezultat al transformării). In felul acesta se poate trece între diferite reprezentări posibile ale documentelor XML, cu sau fãrã modificări în continutul documentelor.

Utilizarea unui parser SAX

Utilizarea unui parser SAX implicã definirea unei clase care implementeazã una sau mai multe interfete (*ContentHandler, DTDHandler, ErrorHandler)* sau care extinde clasa *DefaultHandler* (clasã adaptor care implementeazã toate metodele interfetelor prin functii cu efect nul).

Principalele metode "callback" care trebuie definite în aceastã clasã sunt:

```
// apelata la inceput de element (marcaj de inceput)
public void startElement (String uri, String localName,String qName, Attributes atts)
    throws SAXException;
// apelata la sfarsit de element (marcaj de terminare)
public void endElement (String uri, String localName, String qName)
    throws SAXException;
// apelata la detectarea unui sir de caractere
public void characters (char ch[ ], int start, int length)
    throws SAXException;
```

Metodele activate la întâlnirea de marcaje primesc numele marcajului sub douã forme: numele local (în cadrul spatiului de nume implicit) si numele "qName", calificat cu identificatorul spatiului de nume. Primul argument (uri) aratã unde este definit spatiul de nume al marcajului. In cazul unui singur spatiu de nume se foloseste numai argumentul "qName", care contine numele marcajului ("tag name").

Interfata *ContentHandler* mai contine câteva metode, mai rar folosite, si metode apelate în cazuri de eroare:

"fatalerror" la erori de utilizare marcaje (not well formed)

“error” la erori de validare fatã de DTD (not valid)

Obiectul parser SAX este creat prin apelul unei metode fabricã fie în constructorul clasei “handler”, fie în functia “main” sau într-o altã functie dintr-o altã clasã. Metoda “parse” a clasei “SAXParser” trebuie sã primeascã obiectul ce contine documentul XML (un flux de I/E) si obiectul “handler”, cu metode de tratare a evenimentelor.

Exemplul urmãtor afiseazã numai datele dintr-un document XML (Java 1.4):

```
 // listare caractere dintre marcaje
import org.xml.sax.*;
import org.xml.sax.helpers.DefaultHandler;
import javax.xml.parsers.*;
import java.io.*;
  // clasa handler ptr evenimente SAX
class saxList extends DefaultHandler {
   public saxList (File f) throws Exception {
      SAXParserFactory factory = SAXParserFactory.newInstance();
      SAXParser saxParser = factory.newSAXParser();
      saxParser.parse( f, this);
   }
    // metode de tratare a evenimentelor SAX
   public void startDocument() throws SAXException {
      System.out.println ();
   }
   public void startElement(String uri, String sName, String qName, Attributes at)
                        throws SAXException {
   }
   public void endElement(String uri, String sName, String qName, Attributes at)
                       throws SAXException {
      System.out.println ();
   }
   public void characters(char buf[], int offset, int len) throws SAXException {
      String s = new String(buf, offset, len);
      System.out.print( s.trim());                  // fara spatii nesemnificative
   }
   public static void main(String argv[]) throws Exception {
      DefaultHandler handler = new saxList(new File(argv[0]));  // nume fisier in arg[0]
   }
}
```

Pentru documentul XML anterior cu lista de preturi se va afisa:

```
CDC     540
SDS     495
```

In exemplul urmãtor se creeazã si se afiseazã un arbore JTree pe baza datelor primite de la un parser SAX:

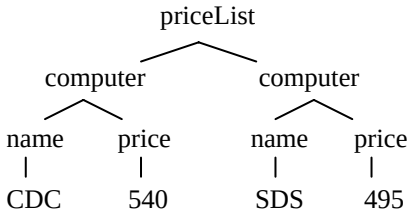
```
import javax.xml.parsers.*;
import java.io.*;
import org.xml.sax.*;
```

```
import org.xml.sax.helpers.*;
import javax.swing.*;
import javax.swing.tree.*;

  // Utilizare parser SAX ptr creare arbore Swing cu continut document XML
class TNode extends DefaultMutableTreeNode {
  public TNode() { super();}
  public TNode (Object info) { super(info);}
}
class SAXTreeViewer extends JFrame {
    private JTree jTree;
    public SAXTreeViewer(String file) throws Exception{
        TNode root = new TNode( file);
        jTree = new JTree(root);
        getContentPane().add(new JScrollPane(jTree));
        DefaultHandler handler =  new JTreeContentHandler(root);
        SAXParserFactory factory = SAXParserFactory.newInstance();
        SAXParser saxParser = factory.newSAXParser();
        saxParser.parse( new File(file), handler);
    }
       // “main” poate fi si intr-o alta clasa
    public static void main(String[] args) throws Exception {
        SAXTreeViewer viewer = new SAXTreeViewer(args[0]);
        viewer.setSize(600, 450);    viewer.show();
    }
}
     // tratare evenimente SAX
class JTreeContentHandler extends DefaultHandler {
    private TNode  crt;                 // nod curent
    public JTreeContentHandler(TNode root) {
        crt = root;
    }
    public void startElement(String nsURI, String sName, String qName,
                                        Attributes atts) throws SAXException {
        TNode element =  new TNode(qName);              // nod cu marcaj
        crt.add(element);               // adauga la nodul curent
        crt = element;                  // noul nod devone nod curent
    }
    public void endElement(String nsURI, String sName, String qName)
                    throws SAXException {
        crt = (TNode)crt.getParent();               // nodul parinte devine nod curent
    }
    public void characters(char[] ch, int start, int length) throws SAXException {
        String s = new String(ch, start, length).trim();
        if (s.length() > 0) {
          TNode data =  new TNode(s);           // creare nod cu text
           crt.add(data);
        }
    }
}
```

Utilizarea unui parser DOM

Pentru fisierul XML anterior cu lista de preturi, dar scris tot pe o singurã linie si fãrã spatii albe, arborele DOM aratã astfel:



De fapt, orice arbore DOM mai are un nivel rãdãcinã (nefigurat aici) care corespunde întregului document XML (nod cu numele “#document” si valoarea *null*).

Fiecare nod dintr-un arbore DOM are un nume, o valoare si un tip. Tipurile sunt numere întregi, dar existã si nume mnemonice pentru aceste tipuri. Exemple de constante din interfata *Node*:

```
public static final short ELEMENT_NODE       = 1;
public static final short ATTRIBUTE_NODE     = 2;
public static final short TEXT_NODE          = 3;
```

Nodurile cu text au toate acelasi nume (“#text”), iar valoarea este sirul de caractere ce reprezintã textul. In exemplul anterior cele 4 noduri terminale sunt noduri text (cu valorile “CDC”,”540”,”SDS”,”495”). Parserul DOM creeazã noduri text si pentru grupuri de spatii albe (blancuri sau terminator de linie înainte sau dupã un marcaj). Pentru fisierul XML cu 10 linii si indentare numãrul de noduri din arborele DOM este 20 : 1 pe nivelul 1, 5 pe nivelul 2, 10 pe nivelul 3 si 4 pe nivelul 4. Dintre aceste noduri sunt 9 noduri text cu spatii albe (3 pe nivelul 2 si 6 pe nivelul 3).

Nodurile pentru elemente au numele marcajului si valoarea *null*. In arborele de mai sus am scris numele nodurilor de pe primele 3 niveluri (de tip 1) si valorile nodurilor de pe ultimul nivel (de tip 3).

Modelul DOM defineste mai multe interfete Java :

- Interfata *Node* contine metode de acces la un nod de arbore DOM si la succesorii sãi, dar si metode pentru crearea si modificarea de noduri de arbore DOM. Exemple:

```
public String getNodeName();          // numele acestui nod
public String getNodeValue() throws DOMException;     // valoarea acestui nod
public void setNodeValue(String nodeValue) throws DOMException;
public short getNodeType();           // tipul acestui nod
public NodeList getChildNodes();      // lista succesorilor acestui nod
public Node getFirstChild();          // primul succesor al acestui nod
public Node getNextSibling();         // urmatorul succesor al acestui nod
public Node removeChild(Node oldChild) throws DOMException;
public Node appendChild(Node newChild) throws DOMException;
```

- Interfata *NodeList* contine metode pentru acces la lista de succesori a unui nod:

```
public Node item(int index);        // nodul cu numarul “index”
public int getLength();             // lungime lista de noduri
```

- Interfetele *Document, Element, Attr, CharacterData, Text, Comment,* extind direct sau indirect interfata *Node* cu metode specifice acestor tipuri de noduri.

Clasele care implementeazã aceste interfete fac parte din parserul DOM, iar numele si implementarea lor nu sunt cunoscute utilizatorilor; ele constituie un bun exemplu de separare a interfetei de implementare si de programare la nivel de interfatã. Clasa care implementeazã interfata *Node* (si extinde implicit clasa *Object*) contine si o metodã “toString”, cu numele si valoarea nodului între paranteze.

Afisarea sau prelucrarea unui arbore DOM se face de obicei printr-o functie staticã recursivã care prelucreazã nodul curent (primit ca argument) si apoi se apeleazã pe ea însãsi pentru fiecare succesor. Parcurgerea listei de succesori se poate face în douã moduri:

- Folosind metodele “getFirstChild” si “getNextSibling” ;
- Folosind metoda “getChildNodes” si metode ale interfetei *NodeList*.

Exemplu pentru prima solutie a functiei recursive:

```
public static void printNode(Node node) {
   System.out.println (node.getNodeName()+"["+ node.getNodeValue()+"]");
   Node  child = node.getFirstChild();      // primul fiu
   while (child !=null) {
     printNode (child);                     // apel recursiv
     child=child.getNextSibling();   // urmatorul frate
   }
}
```

Exemplu pentru a doua solutie a functiei recursive:

```
public static void printNode(Node node) {
   System.out.println ("["+node.getNodeName()+":"+node.getNodeValue()+"]");
   NodeList  kids = node.getChildNodes();       // lista de fii
   if (kids != null)                            // daca exista fii ai nodului curent
     for (int k=0; k<kids.getLength(); k++)     // enumerare fii
       printNode (kids.item(k));                // apel recursiv pentru fiecare fiu
}
```

Dacã operatiile efectuate la fiecare nod depind de tipul nodului, atunci functia recursivã va contine o selectie dupã tipul nodului (un bloc *switch*).

O aplicatie DOM creeazã mai întâi un obiect parser (printr-o fabricã de obiecte) si apoi apeleazã metoda “parse” pentru acest obiect, cu specificarea unui fisier XML.

Metoda “parse” are ca rezultat un obiect de tip *Document*, care este arborele creat pe baza documentului XML, conform modelului DOM.

Exemplul urmãtor foloseste o functie recursivã pentru afisarea tuturor nodurilor dintr-un arbore DOM (cu nume si valoare), pornind de la rãdãcinã.

```
import javax.xml.parsers.*;
import java.io.*;
import org.w3c.dom.*;
  // afisare nume si valori noduri din arbore DOM (Java 1.4)
class DomEcho {
    public static void printNode(Node node) {
       Node child;
       String val= node.getNodeValue();
       System.out.println (node.getNodeName()+"["+ val+"]");
       child = node.getFirstChild();
       if (child !=null) {
         printNode (child);
         while ( (child=child.getNextSibling())!=null)
           printNode(child);
       }
    }
    public static void main (String arg[]) throws Exception {
       DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
       DocumentBuilder builder = factory.newDocumentBuilder();
       Document doc = builder.parse( new File(arg[0]) );
       printNode(doc.getDocumentElement());
    }
}
```

Pentru afisarea unui arbore DOM sub forma unui arbore *JTree* se poate crea un arbore Swing cu datele din arborele DOM, sau se poate stabili o corespondentã nod la nod între cei doi arbori sau se poate folosi o clasã adaptor.

Clasa adaptor primeste apeluri de metode din interfata *TreeeModel* si le transformã în apeluri de metode din interfata *Node* (DOM) . Exemplu:

```
class NodeAdapter implements TreeNode {
   Node node;
   public NodeAdapter(Node node) {
      this.node = node;
   }
   public String toString() {
      return domNode.toString();        //  [nume:valoare]
   }
   public int getIndex(TreeNode child) {   // indicele unui fiu dat
     int count = getChildCount();
     for (int i=0; i<count; i++) {
        NodeAdapter n = (NodeAdapter) (this.getChildAt(i));
        if (((NodeAdapter)child).domNode == n.domNode) return i;
     }
     return -1; // nu ar trebui sa se ajunga aici.
   }
   public TreeNode getChildAt(int i) {     // fiul cu indice dat
     Node node =  domNode.getChildNodes().item(i);
     return new NodeAdapter (node);
   }
```

```
  public int getChildCount() {              // numar de fii
    return domNode.getChildNodes().getLength();
  }
  public TreeNode getParent() {             // parintele acestui nod
    return new NodeAdapter( domNode.getParentNode());
  }
  public boolean isLeaf() {                 // daca acest nod e o frinza
    return getChildCount()==0;
  }
  public boolean getAllowsChildren() {  // daca acest nod poate avea fii
    if (domNode.getNodeType()==Node.ELEMENT_NODE) return true;
    else return false;
  }
  public Enumeration children () {          // enumerator pentru fii acestui nod
    return new CEnum (domNode);
  }
}
```

Clasa enumerator putea fi si o clasã inclusã anonimã:

```
 class CEnum implements Enumeration {
    NodeList list; int i;
    CEnum (Node node) {
       list=node.getChildNodes();
       i=0;
    }
    public Object nextElement () {
      return list.item(i++);
    }
    public boolean hasMoreElements () {
      return i < list.getLength();
    }
  }
}
```

Utilizarea clasei adaptor se va face astfel:

```
public static void main(String argv[ ]) throws Exception {
   DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
   DocumentBuilder builder = factory.newDocumentBuilder();
   Document doc = builder.parse( new File(argv[0]) );
   JFrame frame = new JFrame(argv[0]);
   JTree tree = new JTree(new DefaultTreeModel(new NodeAdapter(doc)));
   frame.getContentPane().add ( new JScrollPane(tree) );
   frame.setSize(500,480);   frame.setVisible(true);
}
```

14. Scheme de proiectare

Proiectarea orientatã pe obiecte

Realizarea unui program cu clase pentru o aplicatie realã necesitã o etapã initialã de proiectare înainte de a trece la scrierea codului. Este posibil ca dupã implementarea proiectului initial sã se revinã la aceastã etapã, fie pentru îmbunãtãtirea proiectului (prin "refactorizare"), fie pentru modificarea sa esentialã.

In faza de proiectare se stabilesc clasele si interfetele necesare si relatiile dintre acestea. O parte din clasele aplicatiei sunt noi si specifice aplicatiei, iar altele sunt clase predefinite, de uz general (colectii de date, fisiere, componente vizuale de interfatã etc.). Aplicatiile Java beneficiazã de un numãr mare de clase de bibliotecã.

Un program si modulele sale componente trebuie proiectate de la început având în vedere posibilitatea de a fi extins si adaptat ulterior într-un mod cât mai simplu si mai sigur (proiectare în perspectiva schimbãrii = “design for change”).

Extindere simplã si sigurã înseamnã cã nu se va recurge la modificarea unor clase existente si se vor defini noi clase, care fie vor înlocui clase din programul existent, fie se vor adãuga claselor existente. In plus, trebuie avutã în vedere si posibilitatea de reutilizare a unor clase din aplicatie si în alte aplicatii.

Clasele si obiectele necesare într-o aplicatie pot rezulta din :

- Analiza aplicatiei concrete si modelarea obiectelor si actiunilor din aplicatie;
- Analiza altor aplicatii si a bibliotecilor de clase existente, care aratã cã pot fi necesare clase cu un caracter mai abstract fie pentru gruparea de obiecte, fie ca intermediar între obiecte, fie cu un alt rol care nu este evident din descrierea aplicatiei.
- Cunoasterea unor solutii de proiectare deja folosite cu succes în alte aplicatii.

In general se poate afirma cã mãrirea flexibilitãtii în extinderea si adaptarea unei aplicatii se poate obtine prin mãrirea numãrului de clase si obiecte din aplicatie. Un proiect ce contine numai clasele rezultate din analiza aplicatiei poate fi mai compact, dar nu este scalabil si adaptabil la alte cerinte apãrute dupã realizarea prototipului aplicatiei. In plus, este importantã o separare a pãrtilor susceptibile de schimbare de pãrtile care nu se mai modificã si evitarea cuplajelor "strânse" dintre clase cooperante.

Pentru un proiect mai complex este utilã crearea de modele pentru pãrti ale aplicatiei, modele care sã reprezinte relatiile dintre clase si obiecte, rolul fiecãrui obiect în aplicatie. Limbajul UML (Unified Modeling Language) este destinat unei descrieri grafice si textuale pentru astfel de modele, într-un mod unitar, iar folosirea sa poate fi utilã pentru proiecte foarte mari, cu numãr mare de clase si obiecte.

Scheme de proiectare

Experienta acumulatã în realizarea unor aplicatii cu clase a condus la recunoasterea si inventarierea unor scheme (sabloane) de proiectare (“Design Patterns”), adicã a unor grupuri de clase si obiecte care cooperează pentru realizarea unor functii. In cadrul acestor scheme existã clase care au un anumit rol în raport cu

alte clase si care au primit un nume ce descrie acest rol; astfel există clase iterator, clase observator, clase “fabricã” de obiecte, clase adaptor s.a. Dincolo de detaliile de implementare se pot identifica clase cu aceleasi responsabilitãti în diferite aplicatii.

Trebuie remarcat cã si clasele predefinite JDK au evoluat de la o versiune la alta în sensul extinderii functionalitãtii si aplicãrii unor scheme de proiectare reutilizabile. Clasele JDK (în special clase JFC) care urmeazã anumite scheme de proiectare au primit nume care sugereazã rolul clasei într-un grup de clase ce interactioneazã : clase iterator, clase adaptor, clase “model” s.a.

Câteva definitii posibile pentru schemele de proiectare folosite în aplicatii cu clase:

- Solutii optime pentru probleme comune de proiectare.
- Reguli pentru realizarea anumitor sarcini în proiectarea programelor cu obiecte.
- Abstractii la un nivel superior claselor, obiectelor sau componentelor.
- Scheme de comunicare (de interactiune) între obiecte.

Argumentul principal în favoarea studierii si aplicãrii schemelor de clase este acela cã aplicarea acestor scheme conduce la programe mai usor de modificat. Aceste obiective pot fi atinse în general prin clase (obiecte) “slab” cuplate, care stiu cât mai putin unele despre altele.

Principalele recomandãri care rezultã din analiza schemelor de proiectare si aplicatiilor sunt:

- Compozitia (agregarea) este preferabilã în raport cu derivarea.
- Proiectarea cu interfete si clase abstracte este preferatã fatã de proiectarea cu clase concrete, pentru cã permite separarea utilizãrii de implementare.
- Este recomandatã crearea de clase si obiecte suplimentare, cu rol de intermediari, pentru decuplarea unor clase cu roluri diferite.

O clasificare uzualã a schemelor de proiectare distinge trei categorii:

- Scheme “creationale“ (Creational patterns) prin care se generează obiectele necesare.
- Scheme structurale (Structural patterns), care grupează mai multe obiecte în structuri mai mari.
- Scheme de interactiune (Behavioral patterns), care definesc comunicarea între clase.

Cea mai folositã schemã de creare obiecte în Java este metoda fabricã, prezentã în mai multe pachete, pentru crearea de diverse obiecte. Se mai foloseste schema pentru crearea de clase cu obiecte unice ("Singleton") în familia claselor colectie.

Schemele structurale folosite in JDK sunt clasele “decorator” din pachetul “java.io” si clasele “adaptor” din “javax.swing”.

Schemele de interactiune prezente în JDK sunt clasele iterator (“java.util”) si clasele observator si observat (“java.util”) extinse în JFC la schema cu trei clase MVC (Model-View-Controller).

Clase si metode “fabricã” de obiecte

In cursul executiei un program creează noi obiecte, care furnizează aplicatiei datele si metodele necesare. Majoritatea obiectelor sunt create folosind operatorul

new, pe baza ipotezei cã existã o singurã clasã, care nu se mai modificã, pentru aceste obiecte.

O solutie mai flexibilã pentru crearea de obiecte este utilizarea unei fabrici de obiecte, care lasã mai multã libertate în detaliile de implementare a unor obiecte cu comportare predeterminatã.

O fabricã de obiecte (“Object Factory”) permite crearea de obiecte de tipuri diferite, dar toate subtipuri ale unui tip comun (interfatã sau clasã abstractã).

O fabricã de obiecte se poate realiza în douã forme:

- Ca metodã fabricã (de obicei metodã staticã) dintr-o clasã, care poate fi clasa abstractã ce defineste tipul comun al obiectelor fabricate. Aceastã solutie se practicã atucni când obiectele fabricate nu diferã mult între ele.
- Ca o clasã fabricã, atunci când existã diferente mai mari între obiectele fabricate.

Alegerea (sub)tipului de obiecte fabricate se poate face fie prin parametri transmisi metodei fabricã sau constructorului de obiecte “fabricã”, fie prin fisiere de proprietãti (fisiere de configurare).

Metode si clase fabricã pot fi întâlnite în diverse subpachete din pachetele “java” sau “javax” (ca parte din Java “Standard Edition” - J2SE) si, mai ales, în clasele J2EE.

Putem deosebi douã situatii în care este necesarã utilizarea unei metode “fabricã” în locul operatorului *new*:

- Pentru a evita crearea de obiecte identice (economie de memorie si de timp); metoda fabricã va furniza la fiecare apel o referintã la un obiect unic si nu va instantia clasa. Rezultatul metodei este de un tip clasã instantiabilã.
- Pentru cã tipul obiectelor ce trebuie create nu este cunoscut exact de programator, dar poate fi dedus din alte informatii furnizate de utilizator, la executie. Rezultatul metodei este de un tip interfatã sau clasã abstractã, care include toate subtipurile de obiecte fabricate de metodã (fabricã "polimorficã").

O metodã fabricã poate fi o metodã staticã sau nestaticã. Metoda fabricã poate fi apelatã direct de programatori sau poate fi apelatã dintr-un constructor, ascunzând utilizatorilor efectul real al cererii pentru un nou obiect.

O metodã fabricã de un tip clasã instantiabilã tine evidenta obiectelor fabricate si, în loc sã producã la fiecare apel un nou obiect, produce referinte la obiecte deja existente. Un exemplu este metoda “createEtchedBorder” din clasa *BorderFactory* care creeazã referinte la un obiect chenar unic, desi se pot crea si obiecte chenar diferite cu *new* :

```
JButton b1 = new JButton (“Etched1”), b2= new JButton(“Etched2”);
    // chenar construit cu “new”
b1.setBorder (new EtchedBorder ());                    // chenar “gravat”
    // chenar construit cu metoda fabricã
Border eb = BorderFactory.createEtchedBorder();
b2.setBorder (eb);
```

Pentru acest fel de metode fabricã existã câteva variante:

- Metoda are ca rezultat o referintã la un obiect creat la încãrcarea clasei:

```
public class BorderFactory {
    private BorderFactory () { }                    // neinstantiabilã
    static final sharedEtchedBorder = new EtchedBorder();
    public static Border createEtchedBorder() {
        return sharedEtchedBorder;
    }
    ...  }            // alte metode fabricã în clasã
```

Clasa *BorderFactory* (pachetul "javax.swing") reuneste mai mai multe metode "fabricã" ce produc obiecte chenar de diferite forme pentru componentele vizuale Swing. Toate obiectele chenar apartin unor clase care respectã interfata comunã *Border* si care sunt fie clase predefinite, fie clase definite de utilizatori. Exemple de clase chenar predefinite: *EmptyBorder, LineBorder, BevelBorder, TitleBorder* si *CompundBorder* (chenar compus).

- Metoda creeazã un obiect numai la primul apel, dupã care nu mai instantiazã clasa.

Un exemplu de metodã fabricã controlabilã prin argumente este metoda "getInstance" din clasa *Calendar*, care poate crea obiecte de diferite subtipuri ale tipului *Calendar* , unele necunoscute la scrierea metodei dar adãugate ulterior.

Obiectele de tip datã calendaristicã au fost create initial în Java ca instante ale clasei *Date*, dar ulterior s-a optat pentru o solutie mai generalã, care sã tinã seama de diferitele tipuri de calendare folosite pe glob. Clasa abstractã *Calendar* (din "java.util") contine câteva metode statice cu numele "getInstance" (cu si fãrã parametri), care fabricã obiecte de tip *Calendar*. Una din clasele instantiabile derivatã din *Calendar* este *GregorianCalendar*, pentru tipul de calendar folosit în Europa si în America. Metoda "getInstance" fãrã argumente produce implicit un obiect din cel mai folosit tip de calendar:

```
public static Calendar getInstance() {
    return new GregorianCalendar();
}
```

Obiectele de tip *Calendar* se pot utiliza direct sau transformate în obiecte *Date*:

```
Calendar azi = Calendar.getInstance();
Date now = azi.getTime();          // conversie din Calendar in Date
System.out.println (now);
```

Metoda "getInstance" poate avea unul sau doi parametri, unul de tip *TimeZone* si altul de tip *Local*, pentru adaptarea orei curente la fusul orar (TimeZone) si a calendarului la pozitia geograficã (Local). Tipul obiectelor fabricate este determinat de acesti parametri.

Alte metode fabricã care produc obiecte adaptate particularitatilor locale (de tarã) se aflã în clase din pachetul "java.text": *NumberFormat, DateFormat*, s.a. Adaptarea se face printr-un parametru al metodei fabricã care precizeazã tara si este o constantã simbolicã din clasa *Locale*.

Afisarea unei date calendaristice se poate face în mai multe forme, care depind de uzantele locale si de stilul dorit de utilizator (cu sau fãrã numele zilei din sãptãmânã,

cu nume complet sau prescurtat pentru lunã etc.). Clasa *DateFormat* contine metoda "getDateInstance" care poate avea 0,1 sau 2 parametri si care poate produce diferite obiecte de tip *DateFormat*. Exemple:

```
Date date = new Date();
DateFormat df1 = DateFormat.getDateInstance ();
DateFormat df2 = DateFormat.getDateInstance (2, Locale.FRENCH);
System.out.println ( df1.format(date));      //luna, an, zi cu luna in engleza
System.out.println ( df2.format(date));      // zi,luna,an cu luna in franceza
```

Fabrici abstracte

Clasele de obiecte fabricã pot fi la rândul lor subtipuri ale unui tip comun numit fabricã abstractã ("AbstractFactory"). Altfel spus, se folosesc uneori fabrici de fabrici de obiecte (frecvent în J2EE – Java 2 Enterprise Edition).

Uneori se stie ce fel de obiecte sunt necesare, dar nu se stie exact cum trebuie realizate aceste obiecte (cum trebuie implementatã clasa); fie nu existã o singurã clasã posibilã pentru aceste obiecte, fie existau anterior furnizori diferiti pentru aceste clase si se încearcã unificarea lor, fie sunt anticipate si alte implementãri ale unei clase în viitor, dar care sã nu afecteze prea mult aplicatiile existente.

Portabilitatea doritã de Java cere ca operatii (cereri de servicii) sã fie programate uniform, folosind o singurã interfatã API între client (cel care solicitã operatia) si furnizorul de servicii ("provider"), indiferent de modul cum sunt implementate acele servicii, sau de faptul cã existã mai multe implementãri.

Astfel de interfete API existã pentru servicii prin natura lor diversificate ca mod de realizare: pentru acces la orice bazã de date relationalã (JDBC), pentru comunicarea prin mesaje (JMS, SAAJ), pentru analizã de fisiere XML etc.

Pentru acces la servicii oferite de un server se creeazã un obiect "conexiune", iar operatiile ulterioare se exprimã prin apeluri de metode ale obiectului "conexiune". Diversitatea apare în modul de creare a obiectului conexiune, care nu se poate face prin instantierea unei singure clase, cu nume si implementare cunoscute la scrierea aplicatiei. Obiectele conexiune sunt create de fabrici de conexiuni, care ascund particularitãti de implementare ale acestor obiecte. Toate obiectele conexiune trebuie sã prevadã anumite metode, deci sã respecte un contract cu utilizatorii de conexiuni. Fabricile de conexiuni pot fi uneori si ele foarte diferite si sã necesite parametri diferiti (ca tip si ca utilizare); de aceea, obiectul fabricã de conexiuni nu este obtinut prin instantierea unei singure clase ci este fabricat într-un fel sau altul.

In JDBC clasa fabricã de conexiuni, numitã "DriverManager" foloseste o clasã "driver", livratã de obicei de cãtre furnizorul bazei de date, sub forma unei clase cu nume cunoscut (încãrcatã înainte de utilizarea clasei fabricã) . Exemplu:

```
Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");   // clasa driver
Connection con = DriverManager.getConnection (dbURL,user,passwd);
Statement stm = con.createStatement();    // utilizare obiect conexiune
```

In exemplul anterior *Connection* si *Statement* sunt interfete, dar nu există în pachetul “java.sql” nici o clasã instantiabilã care sã implementeze aceste interfete. Obiectul conexiune este fabricat de metoda “getConnection” pe baza informatiilor extrase din clasa driver; la fel este fabricat si obiectul “instructiune” de cãtre metoda “createStatement”.

Exemplu de utilizare a unei fabrici de fabrici de conexiuni pentru mesaje SOAP:

```
SOAPConnectionFactory factory = SOAPConnectionFactory.newInstance();
SOAPConnection con = factory.createConnection();
…
```

In exemplul anterior obiectul fabricã “factory” este produs de o fabricã abstractã implicitã (numele este ascuns în metoda “newInstance”), furnizatã de firma “Sun” pentru a permite exersarea de mesaje SOAP, fãrã a exclude posibilitatea altor fabrici de conexiuni SOAP, de la diversi alti furnizori.

Pentru a întelege mai bine ce se întâmplã într-o fabricã (de fabrici) de obiecte am imaginat urmãtorul exemplu: o fabricã pentru crearea de obiecte colectie (de orice subtip al tipului *Collection*) pe baza unui vector implicit de obiecte. Exemplu:

```
public static Collection newCollection (Object a[], Class cls)  {
  Collection c=null;
  try {  c = (Collection) cls.newInstance(); }
  catch (Exception e) { System.out.println(e);}
  for (int i=0;i<a.length;i++)
      c.add (a[i]);
   return c;
  }
        // utilizare (in “main”)
    String a[ ] = { "1","2","3","2","1"};
    HashSet set = (HashSet) newCollection (a,HashSet.class);
```

In exemplul anterior am folosit o metodã fabricã de obiecte colectie, care va putea fi folositã si pentru clase colectie încã nedefinite, dar care implementeazã interfata. Utilizarea acestei fabrici este limitatã deoarece nu permite transmiterea de parametri la crearea obiectelor colectie; de exemplu, la crearea unei colectii ordonate este necesarã transmiterea unui obiect comparator (subtip al tipului *Comparator*).

O solutie ar putea fi definirea mai multor clase fabricã de colectii, toate compatibile cu un tip comun (tipul interfatã “CollectionFactory”). Exemplu:

```
  // interfata pentru fabrici de colectii
interface CollectionFactory {
  public Collection newCollection (Object [] a);
}
  // fabrica de colectii simple (neordonate)
class SimpleCollectionFactory implements CollectionFactory {
  private Collection col;
  public SimpleCollectionFactory (String clsName) {
    try {
```

```
      Class cls = Class.forName(clsName);
      col =(Collection) cls.newInstance();
    } catch (Exception e) { System.out.println(e);}
  }
  public Collection newCollection (Object a[]) {
    for (int i=0;i<a.length;i++)
       col.add (a[i]);
    return col;
}
}
  // fabrica de colectii ordonate
class SortedCollectionFactory implements CollectionFactory {
  private Collection col;
  public SortedCollectionFactory (String clsName, Comparator comp) {
     try {
      Class cls = Class.forName(clsName);
      Class[] clsargs = new Class[] {Comparator.class};
      Object[] args = new Object[] { comp };
      Constructor  constr = cls.getConstructor(clsargs);
      col =(Collection) constr.newInstance(args);
    } catch (Exception e) { System.out.println(e);}
  }
  public Collection newCollection (Object a[]) {
    for (int i=0;i<a.length;i++)
       col.add (a[i]);
    return col;
}
```

Instantierea directã a acestor clase fabricã ar obliga utilizatorul sã cunoascã numele lor si ar necesita modificări în codul sursã pentru adãugarea unor noi clase fabricã. De aceea, vom defini o metodã staticã (supradefinitã) cu rol de fabricã de obiecte fabricã:

```
     // metode fabrica de fabrici de colectii
class FFactory {
  public static CollectionFactory getFactory (String clsName) {
    return new SimpleCollectionFactory (clsName);
  }
  public static CollectionFactory getFactory (String clsName,Comparator comp) {
    return new SortedCollectionFactory (clsName,comp) ;
  }
}
     // exemple de utilizare
public static void main (String arg[]) throws Exception {
    String a[] = { "1","2","3","2","1"};
    CollectionFactory cf1 = FFactory.getFactory ("java.util.HashSet");
    HashSet set1 = (HashSet) cf1.newCollection (a);
    CollectionFactory cf2 = FFactory.getFactory ("java.util.TreeSet", new MyComp());
    TreeSet set2 = (TreeSet) cf2.newCollection (a);
```

```
   System.out.println(set1);  System.out.println(set2);
}
```

In acest caz particular se putea folosi o singurã clasã fabricã de colectii cu mai multi constructori, deoarece sunt diferente mici între cele douã fabrici. In situatii reale pot fi diferente foarte mari între clasele fabricã pentru obiecte complexe.

Clase cu rol de comandã (actiune)

Schema numitã "Command" permite realizarea de actiuni (comenzi) multiple si realizeazã decuplarea alegerii operatiei executate de locul unde este emisã comanda. Ea foloseste o interfatã generalã, cu o singurã functie, de felul urmãtor:

```
public interface Command {
    public void execute();          // executã o actiune nedefinitã încã
}
```

In Swing existã mai multe interfete "ascultãtor" corespunzãtoare interfetei “Command”. Exemplu:

```
public interface ActionListener {
    public void actionPerformed( ActionEvent ev);
}
```

Un obiect dintr-o clasã ascultãtor este un obiect "comandã" si realizeazã o actiune. El constituie singura legãturã dintre partea de interfatã graficã a aplicatiei si partea de logicã specificã aplicatiei (tratarea evenimentelor). Se realizeazã astfel decuplarea celor douã pãrti , ceea ce permite modificarea lor separatã si chiar selectarea unei anumite actiuni în cursul executiei.

Pentru a compara solutia schemei "Command" cu alte solutii de selectare a unei actiuni (comenzi), sã examinãm pe scurt programarea unui meniu cu clase JFC. Un meniu este format dintr-o barã meniu orizontalã (obiect *JMenuBar*), care contine mai multe optiuni de meniu (obiecte *JMenu*) si fiecare optiune poate avea un submeniu vertical cu mai multe elemente de meniu (obiecte *JMenuItem*). Un element de meniu este o optiune care declanseazã o actiune la selectarea ei, prin producerea unui eveniment de tip *ActionEvent*.

Pentru programarea unui meniu cu o singurã optiune "File", care se extinde într-un meniu vertical cu douã elemente ("Open" si "Exit") sunt necesare instructiunile:

```
JMenuBar mbar = new JMenuBar();         // creare bara meniu principal
setJMenuBar (mbar);                     // adauga bara meniu la JFrame
JMenu mFile = new JMenu ("File");       // optiune pe bara orizontala
mbar.add (mFile);                       // adauga optiunea mFile la bara meniu
JMenuItem open = new JMenuItem ("Open");// optiune meniu vertical
JMenuItem exit = new JMenuItem ("Exit");    // optiune meniu vertical
mFile.add (open); mFile.addSeparator( );    // adauga optiuni la meniu vertical
mFile.add (exit);
```

```
open.addActionListener (this);              // sau un alt obiect ca argument
exit.addActionListener (this);              // sau un alt obiect ca argument
```

Tratarea evenimentelor generate de diverse elemente de meniu (mai multe surse de evenimente, în aplicatiile reale) se poate face într-o singurã metodã:

```
public void actionPerformed (ActionEvent e) {
  Object source = e.getSource();            // sursa evenimentului
  if ( source == open)  fileOpen( );        // actiune asociata optiunii "Open"
  else if (source == exit) System.exit(0);  // actiune asociata optiunii "Exit"
}
```

Aplicarea schemei "Command", cu o interfatã "Command", conduce la definirea mai multor clase care implementeazã aceastã interfatã (clase de nivel superior sau clase incluse în clasa *JFrame*, pentru acces la alte variabile din interfata graficã). Exemplu:

```
class FileExitCmd  extends JMenuItem implements Command {
    public FileExitCmd (String optname) {
      super(optname);                       // nume optiune (afisat in meniu)
    }
    public void execute () {
      System.exit(0);                       // actiune asociata optiunii
    }
}
```

In programarea anterioarã a meniului apar urmãtoarele modificãri:

```
class CmdListener implements ActionListener {
  public void actionPerformed (ActionEvent e) {
      Commnad c = (Command) e.getSource();
       c.execute();
  }
}
. . .
ActionListener cmd = new CmdListener();
open.addActionListener (cmd);
exit.addActionListener (cmd);
```

Metoda "execute" este o metodã polimorficã, iar selectarea unei implementãri sau alta se face în functie de tipul variabilei “c”, deci de tipul obiectului care este sursa evenimentului.

Schema "Command" mai este folositã în programarea meniurilor si barelor de instrumente ("toolbar"), prin utilizarea obiectelor "actiune", care implementeazã interfata *Action*. O barã de instrumente contine mai multe butoane cu imagini (obiecte *JButton*), dar are acelasi rol cu o barã meniu: selectarea de actiuni de cãtre operator.

In loc sã se adauge celor douã bare obiecte diferite (dar care produc aceeasi actiune) se adaugã un singur obiect, de tip *Action*, care contine o metodã

"actionPerformed"; câte un obiect pentru fiecare actiune selectatã. Interfata *Action* corespunde interfetei "Command", iar metoda "actionPerformed" corespunde metodei "execute".

Interfata *Action* extinde interfata *ActionPerformed* cu douã metode "setValue" si "getValue", necesare pentru stabilirea si obtinerea proprietãtilor unui obiect actiune: text afisat în optiune meniu, imagine afisatã pe buton din bara de instrumente, o scurtã descriere ("tooltip") a "instrumentului" afisat.

Clase cu un singur obiect

Uneori este nevoie de un singur obiect de un anumit tip, deci trebuie interzisã instantierea repetatã a clasei respective, numitã clasã "Singleton". Existã câteva solutii:

- O clasã fãrã constructor public, cu o metodã staticã care produce o referintã la unicul obiect posibil (o variabilã staticã si finalã contine aceastã referintã).
- O clasã staticã inclusã si o metodã staticã (în aceeasi clasã exterioarã) care instantiazã clasa (pentru a-i transmite un parametru).

Ambele solutii pot fi întâlnite în clasa *Collections* (neinstantiabilã) pentru a se crea obiecte din colectii "speciale" : colectii vide *(EmptySet, EmptyList, EmptyMap)* si colectii cu un singur obiect (*SingletonSet*, *SingletonList*, *SingletonMap*). Exemplu:

```
public class Collections {
    public static Set singleton(Object o) {
     return new SingletonSet(o);
    }
   private static class SingletonSet extends AbstractSet  {
        private Object element;
        SingletonSet(Object o) {element = o;}
        public int size() {return 1;}
        public boolean contains(Object o) {return eq(o, element);}
        ...   // public Iterator iterator() { ... }
    }
    . . .
}
```

Exemplu de utilizare în problema claselor de echivalentã (componente conexe graf):

```
public class Sets {                    // colectie de multimi disjuncte
  private List sets;                   // lista de multimi
    // constructor ( n= nr de elemente in colectie)
  public Sets (int n) {
    sets = new ArrayList (n);
    for (int i=0;i<n;i++)
      sets.add (new TreeSet (Collections.singleton ( new Integer(i) ) ));
  }
  . . .  // metode find, union
```

```
}
```

Schema claselor observat – observator

In aceastã schemã existã un obiect care poate suferi diverse modificãri (subiectul observat) si unul sau mai multe obiecte observator, care ar trebui anuntate imediat de orice modificare în obiectul observat, pentru a realiza anumite actiuni.

Obiectul observat contine o listã de referinte la obiecte observator si, la producerea unui eveniment, apeleazã o anumitã metodã a obiectelor observator înregistrate la el.

Relatia dintre o clasã JFC generator de evenimente si o clasã ascultãtor (receptor) care reactioneazã la evenimente este similarã relatiei dintre o clasã observatã si o clasã observator. De exemplu, un buton *JButton* are rolul de obiect observat, iar o clasã care implementeazã interfata *ActionListener* si contine metoda “actionPerformed” are rolul de observator.

Schema observat-observator a generat clasa *Observable* si interfata *Observer* din pachetul "java.util". Programatorul de aplicatie va defini una sau mai multe clase cu rol de observator, compatibile cu interfata *Observer*. Motivul existentei acestei interfete este acela cã în clasa *Observable* (pentru obiecte observate) existã metode cu argument de tip *Observer* pentru mentinerea unui vector de obiecte observator. Exemplu:

```
public void addObserver(Observer o) {      // adauga un nou observator la lista
  if (!observers.contains(o))         // "observers" este vector din clasa Observable
    observers.addElement(o);
  }
```

Interfata *Observer* contine o singurã metodã "update", apelatã de un obiect observat la o schimbare în starea sa care poate interesa obiectele observator înregistrate anterior:

```
public interface Observer {
   void update(Observable o, Object arg);       // o este obiectul observat
}
```

Clasa *Observable* nu este abstractã dar nici nu este direct utilizabilã; programatorul de aplicatie va defini o subclasã care preia toate metodele clasei *Observable*, dar adaugã o serie de metode specifice aplicatiei care apeleazã metodele superclasei “setChanged” si “notifyObservers”. Metoda “notifyObservers” apeleazã metoda "update" pentru toti observatorii introdusi în vectorul "observers":

```
public class Observable {
   private boolean changed = false;              // daca s-a produs o scimbare
   private Vector obs;                           // lista de obiecte observator
   public void addObserver(Observer o) {       // adauga un observator la lista
     if (!obs.contains(o))
        obs.addElement(o);
   }
```

```
   public void notifyObservers(Object arg) {    // notificare observatori de schimbare
     if (!changed)
        return;
     for (int i = arrLocal.length-1; i>=0; i--)
         ((Observer)obs.elementAt(I)).update(this, arg);
   }
   protected void setChanged() {          // comanda modificare stare ob. observat
     changed = true;
   }
 . . .
```

Exemplu de definire a unei subclase pentru obiecte observate:

```
class MyObservable extends Observable {
    public void change() {       // metoda adagata in subclasa
        setChanged();            // din clasa Observable
        notifyObservers();       // anunta observatorii ca s-a produs o schimbare
    }
}
```

Clasa observator poate arãta astfel, dacã metoda “update” adaugã câte un caracter la o barã afisatã pe ecran (la fiecare apelare de cãtre un obiect MyObservable).

```
class ProgressBar implements Observer {
    public void update( Observable obs, Object x ) {
        System.out.print('#');
    }
}
```

Exemplu de utilizare a obiectelor observat-observator definite anterior:

```
public static void main(String[ ] av) {
    ProgressBar bar= new ProgressBar();
    MyObservable model = new MyObservable();
    model.addObserver(bar);
      int n=1000000000, m=n/100;
      for (int i=0;i<n;i++)         // modifica periodic stare obiect observat
        if ( i%m==0)
          model.change();
  }
```

Interfetele *ActionListener, ItemListener* s.a. au un rol similar cu interfata *Observer,* iar metodele "actionPerformed" s.a. corespund metodei "update". Programatorul de aplicatie defineste clase ascultãtor compatibile cu interfetele "xxxListener", clase care implementeazã metodele de tratare a evenimentelor ("actionPerformed" s.a).

Corespondenta dintre metodele celor douã grupe de clase este astfel:

addObserver	addActionListener, addChangeListener
notifyObservers	fireActionEvent, fireChangeEvent
update	actionPerformed, stateChanged

Clase "model" în schema MVC

Schema MVC (Model-View-Controller) este o extindere a schemei "Observat-observator". Un obiect "model" este un obiect observat (ascultat), care generează evenimente pentru obiectele receptor înregistrate la model, dar evenimentele nu sunt produse ca urmare directă a unor cauze externe programului (nu sunt cauzate direct de actiuni ale operatorului uman).

Arhitectura MVC foloseste clase având trei roluri principale:

- Clase cu rol de "model", adică de obiect observat care contine si date.
- Clase cu rol de redare vizuală a modelului, adică de obiect observator al modelului.
- Clase cu rol de comandă a unor modificări în model.

Schema MVC a apărut în legătură cu programele de birotică (pentru calcul tabelar si pentru editare de texte), de unde si numele de "Document-View-Controller".

Separarea netă a celor trei componente (M, V si C) permite mai multă flexibilitate în adaptarea si extinderea programelor, prin usurinta de modificare separată a fiecărei părti (în raport cu un program monolit la care legăturile dintre părti nu sunt explicite). Astfel, putem modifica structurile de date folosite în memorarea foii de calcul (modelul) fără ca aceste modificări să afecteze partea de prezentare, sau putem adăuga noi forme de prezentare a datelor continute în foaia de calcul sau noi forme de interactiune cu operatorul (de exemplu, o bară de instrumente "tool bar").

Separarea de responsabilităti oferită de modelul MVC este utilă pentru realizarea de aplicatii sau părti de aplicatii: componente GUI, aplicatii de tip "client-server" s.a.

Pentru a ilustra folosirea schemei MVC în proiectarea unei aplicatii si afirmatia că introducerea de obiecte (si clase) suplimentare face o aplicatie mai usor de modificat vom considera următoarea problemă simplă:

Operatorul introduce un nume de fisier, aplicatia verifică existenta acelui fisier în directorul curent si adaugă numele fisierului la o listă de fisiere afisată într-o altă fereastră. Ulterior vom adăuga o a treia fereastră în care se afisează dimensiunea totală a fisierelor selectate. Dacă nu există fisier cu numele introdus atunci se emite un semnal sonor si nu se modifică lista de fisiere (si nici dimensiunea totală a fisierelor).

Obiectele folosite de aplicatie pot fi : *JTextField* pentru introducerea unui nume de fisier, *JTextField* pentru afisare dimensiune totală si *JTextArea* pentru lista de fisiere. Ele vor avea atasate si etichete *JLabel* care explică semnificatia ferestrei. Pentru a avea mai multă libertate în plasarea ferestrelor pe ecran vom crea două panouri.

Fereastra de introducere are rolul de "controler" deoarece comandă modificarea continutului celorlalte două ferestre, care sunt două "imagini" diferite asupra listei de fisiere selectate.

Secventa următoare realizează crearea obiectelor necesare si gruparea lor si este comună pentru toate variantele discutate pentru această aplicatie.

```
class MVC extends JFrame {
  TextArea listView = new TextArea(10,20);      // imagine lista
  TextField sizeView = new TextField(15);       // imagine dimensiune totala
  TextField controler = new TextField(10);      // Controler
  public MVC () {                               // constructor
    Container c = getContentPane();
    c.setLayout (new FlowLayout ());
        // Afisare controler
    JPanel panel1 = new JPanel();               // panou pentru controler si sizeView
    panel1.setLayout(new FlowLayout());
    panel1.add(new JLabel("File")); panel1.add(controler);
    panel1.add( new JLabel("Size"));  panel1.add (sizeView);
    c.add(panel1);
        // Afisare imagini
    JPanel panel2 = new JPanel();                       // panou pentru listView
    panel2.setLayout( new FlowLayout());
    panel2.add( new JLabel("List"));
    panel2.add(new JScrollPane(listView));              // pentru defilare in zona text
    c.add(panel2);
        // Legare imagini la controler
      . . .
  }
  public static void main(String args[]) {
    MVC frame = new MVC();
    frame.setSize(300,200); frame.setVisible(true);    // afisare fereastra
  }
}
```

Programul este utilizabil, dar poate fi extins pentru “slefuirea” aspectului ferestrei principale cu chenare pe componente, cu intervale de separare (“strouts”) si între ferestre si cu o altã grupare a componentelor atomice Swing în panouri.

In varianta aplicatiei cu numai douã ferestre (cu un singur obiect “imagine”) legarea obiectului “listView” la obiectul “controler” se face prin înregistrarea unui obiect ascultãtor la sursa de evenimente de tip *ActionEvent* si prin implementarea metodei “actionPerformed” cu actiunea declansatã în obiectul receptor:

```
        // Legare  listView la controler
controler.addActionListener( new ActionListener() {
  public void  actionPerformed(ActionEvent ev) {
     String newElem = controler.getText();          // sirul introdus in  “controler”
     File f = new File (newElem);                   // ptr a verifica existenta fisierului
     if ( f.exists())
       listView.append ( " "+ newElem +"\n");       // adauga nume la lista afisata
     else
       Toolkit.getDefaultToolkit().beep();          // daca nu exista fisier
     controler.setText("");                         // sterge cimp de intrare
    }
});
```

In secventa anterioarã s-a definit o clasã inclusã anonimã cu rol de “adaptor” între controler si imagine. Continutul clasei adaptor depinde si de tipul componentei de comandã (aici un câmp text) si de rolul componentei imagine în aplicatie.

Pentru extinderea acestei aplicatii cu afisarea dimensiunii totale a fisierelor introduse pânã la un moment dat avem mai multe solutii:

a) Adãugarea unui nou receptor la “controler” printr-o altã clasã adaptor între controler si noul obiect “imagine”. De remarcat cã va trebui sã repetãm o serie de operatii din adaptorul deja existent, cum ar fi verificarea existentei fisierului cu numele introdus în câmpul text. Mai neplãcut este faptul cã succesiunea în timp a executiei metodelor “actionPerformed” din cele douã obiecte adaptor (receptor) nu este previzibilã. De fapt aplicatia functioneazã corect numai dacã este adãugat mai întâi adaptorul pentru câmpul text “sizeView” si apoi adaptorul pentru zona text “listView” (ordinea inversã modificã comportarea aplicatiei):

```
        // Legare controler la sizeView
controler.addActionListener( new ActionListener() {
  int sum=0;
    public void  actionPerformed(ActionEvent ev) {
      String newElem = controler.getText();
      File f = new File (newElem);
      if ( f.exists()) {
        sum+= f.length(); sizeView.setText(" "+ sum);
      }
      else
        Toolkit.getDefaultToolkit().beep();
    }
});
    // Legare controler la listView
controler.addActionListener( new ActionListener() {
  public void actionPerformed(ActionEvent ev) {
    String newElem = controler.getText();
    File f = new File (newElem);
      if ( f.exists()) {
          listView.append ( " "+ newElem +"\n");
      controler.setText("");                          // sterge camp de intrare
  }
});
```

b) Modificarea adaptorului astfel ca sã lege obiectul “controler” la ambele obiecte imagine, actualizate într-o singurã functie apelatã la declansarea evenimentului de modificare a continutului câmpului text “controler”:

```
        // Legare controler la sizeView si listView
 controler.addActionListener( new ActionListener() {
   int sum=0;                                          // suma dimensiuni fisiere
     public void  actionPerformed(ActionEvent ev) {
       String newElem = controler.getText();         // sirul introdus in  “controler”
       File f = new File (newElem);                  // ptr a verifica existenta fisierului
```

```
        if ( f.exists()) {
            sum+= f.length();
            sizeView.setText(" "+ sum);                 // afisare dimensiune totala
          listView.append ( " "+ newElem +"\n");        // adauga nume la lista afisata
        }
        else
            Toolkit.getDefaultToolkit().beep();
       controler.setText("");                           // sterge cimp de intrare
    }
});
```

Desi este forma cea mai compactã pentru aplicatia propusã, modificarea sau extinderea ei necesitã modificări într-o clasã existentã (clasa adaptor), clasã care poate deveni foarte mare dacã numãrul obiectelor ce comunicã si/sau functiile acestora cresc. Din acest motiv clasa adaptor este mai bine sã fie o clasã cu nume, definitã explicit, chiar dacã rãmâne inclusã în clasa MVC.

Dintre modificãrile posibile mentionãm : adãugarea unor noi obiecte imagine (de exemplu o fereastrã cu numãrul de fisiere selectate si o barã care sã arate proportia de fisiere selectate din fisierul curent) si înlocuirea câmpului text din obiectul controler cu o listã de selectie (ceea ce este preferabil, dar va genera alte evenimente si va necesita alte metode de preluare a datelor).

c) Cuplarea obiectului “sizeView” la obiectul “listView” sau invers nu este posibilã, pentru cã un câmp text sau o zonã text nu genereazã evenimente la modificarea lor prin program ci numai la interventia operatorului (evenimentele sunt externe si asincrone programului în executie).

In variantele prezentate aplicatia nu dispune de datele introduse într-un obiect accesibil celorlalte pãrti din aplicatie, desi lista de fisiere ar putea fi necesarã si pentru alte operatii decât afisarea sa (de exemplu, pentru comprimare si adãugare la un fisier arhivã). Extragerea datelor direct dintr-o componentã vizualã Swing nu este totdeauna posibilã si oricum ar fi diferitã pentru fiecare tip de componentã.

Introducerea unui nou obiect cu rol de “model” care sã continã datele aplicatiei (lista de fisiere validate) permite separarea celor trei pãrti din aplicatie si modificarea lor separatã. Pentru problema datã se poate folosi modelul de listã *DefaultListModel*, care contine un vector (si suportã toate metodele clasei *Vector*), dar în plus poate genera trei tipuri de evenimente: adãugare element, eliminare element si modificare continut. Sau, putem defini o altã clasã care implementeazã interfata *ListModel* .

Un receptor al evenimentelor generate de un model *ListModel* trebuie sã implementeze interfata *ListDataListener*, care contine trei metode : “intervalAdded”, “intervalRemoved” si “contentsChanged”. Se poate defini o clasã adaptor “ListDataAdapter” cu toate metodele interfetei *ListDataListener*, dar fãrã nici un efect.

Acum sunt necesare trei clase adaptor care sã lege la model obiectul controler si cele douã obiecte imagine, dintre care douã extind clasa “ListDataAdapter”:

```
DefaultListModel model = new DefaultListModel();        // obiect “model”
. . .
```

```
    // adaptor intre controler si model
class CMAdapter implements ActionListener {
       public void actionPerformed(ActionEvent ev) {
      String file = controler.getText();
      File f= new File(file);
      if (f.exists())
           model.addElement ( file );
      else
        Toolkit.getDefaultToolkit().beep();
      controler.setText(""); // sterge cimp de intrare
    }
  }
    // clase adaptor intre model si imaginile sale
class MV1Adapter extends ListDataAdapter {
       long sum=0; String file; File f;
    public void intervalAdded ( ListDataEvent ev) {
       file = (String) model.lastElement();
       f= new File(file);
       sum += f.length();
       sumView.setText(" "+ sum);
    }
 }
 class MV2Adapter extends ListDataAdapter {
    public void intervalAdded (ListDataEvent ev) {
        listView.append(" "+ model.lastElement()+"\n");
    }
  }
    . . .
    // Cuplare obiecte MVC prin obiecte adaptor
   controler.addActionListener( new CMAdapter());  // model la controler
   model.addListDataListener( new MV1Adapter()); // sumView la model
   model.addListDataListener( new MV2Adapter()); // listView la model
 }
```

O clasã "adaptor" are rolul de adaptare a unui grup de metode (o "interfatã") la un alt grup de metode si reprezintã o altã schemã de proiectare.

Desi textul sursã al aplicatiei a crescut substantial fatã de varianta monolit, extinderea sau modificarea aplicatiei a devenit mai simplã si mai sigurã pentru cã se face prin prin adãugarea de noi obiecte sau prin înlocuirea unor obiecte, fãrã a modifica obiectele existente sau metodele acestor obiecte.

Aplicatia se poate îmbunãtãti prin modificarea modului de alegere a fisierelor de cãtre operator: în loc ca acesta sã introducã mai multe nume de fisiere (cu greselile inerente) este preferabil ca aplicatia sã afiseze continutul unui director specificat si din lista de fisiere operatorul sã le selecteze pe cele dorite. In acest caz se vor mai folosi: un câmp text pentru introducere nume director si un obiect *JList* pentru afisarea listei de fisiere si selectia unor elemente din aceastã listã (prin clic pe mouse).

Refactorizare în programarea cu obiecte

Pentru o aplicatie cu clase există întotdeauna mai multe solutii echivalente ca efect exterior, dar diferite prin structura internă a programelor. In cazul aplicatiilor cu interfată grafică există un număr mare de obiecte care comunică între ele (printre care si obiecte receptor la evenimente generate de diverse componente vizuale) si deci posibilitătile de alegere a structurii de clase sunt mai numeroase.

Ideea de "refactorizare" a programelor cu obiecte porneste de la existenta mai multor variante de organizare a unei aplicatii (de descompunere a aplicatiei în "factori") si sugerează că analiza primei variante corecte poate conduce la alte solutii de organizare a programului, cu aceeasi comportare exterioară dar preferabile din punct de vedere al extinderii si adaptării aplicatiei, sau al performantelor.

Refactorizarea necesită cunoasterea schemelor de proiectare ("Design Patterns"), care pot furniza solutii pentru evitarea cuplajelor "strânse" dintre clase si obiecte.

Anumite medii integrate pentru dezvoltare de aplicatii Java facilitează refactorizarea prin modificări automate în codul sursă.

Codul generat de un mediu integrat pentru scheletul unei aplicatii cu interfată grafică separă net partea fixă (nemodificabilă) de partea ce trebuie completată de utilizator. Exemplu de cod generat de NetBeans pentru o clasă cu un buton:

```
public class gui1 extends javax.swing.JFrame {
   private javax.swing.JButton okButton;
   public gui1() { initComponents();  }
   private void initComponents() {
      okButton = new javax.swing.JButton();   okButton.setText("OK");
      okButton.addActionListener(new java.awt.event.ActionListener() {
         public void actionPerformed(java.awt.event.ActionEvent evt) {
            okButtonActionPerformed(evt);    // dependenta de aplicatie
         }
      });
      …
    }
    private void okButtonActionPerformed(java.awt.event.ActionEvent evt) {
        // cod specific aplicatiei
   }
    public static void main(String args[]) {
       new gui1().setVisible(true);
     }
  }
```

Anexa A. Exemplu de Framework: JUnit

Specificatii

JUnit este numele unei biblioteci de clase Java destinatã scrierii de teste pentru unitãti functionale, deci pentru verificarea claselor (metodelor) scrise sau care urmeazã a fi scrise pentru a fi incluse într-o aplicatie. Testarea codului este necesarã atât în faza de elaborare cât si în faza de depanare a programelor cu erori sau suspectate de erori.

Testarea programelor se face în mod traditional fie prin inserarea unor instructiuni de afisare a unor variabile de control, fie cu ajutorul unui program depanator (debugger), care permite trasarea executiei si afisarea unor expresii de control.

O viziune mai nouã (parte din abordarea numitã XP= Extreme Programming) cere ca orice metodã sã fie însotitã de o functie de test, care poate fi scrisã chiar înainte de implementarea metodei testate. Aceastã practicã permite precizarea conditiilor de functionare corectã a oricãrei metode si posibilitatea de reutilizare a testelor în orice moment, si dupã efectuarea unor modificãri în cod. Un program însotit de teste cât mai complete poate reduce timpul de punere la punct, timpul de întretinere si mãreste încrederea în rezultatele sale.

O metodã de testare a unei metode “f” apeleazã metoda “f” cu anumite date si comparã rezultatul executiei cu valoarea asteptatã.

Pentru exemplificare vom considera verificarea câtorva metode dintr-o clasã care existã în SDK: clasa “BitSet” din pachetul “java.util”. Metoda “set (int x)” pune pe 1 bitul numãrul x dintr-o multime de biti, iar metoda “get(int x)” are rezultat “true” dacã bitul x este 1. Vom presupune cã metoda “set” a fost verificatã fãrã a folosi metoda “get”, astfel ca la verificarea lui “get” sã putem folosi pe “set”.

In cazul în care rezultatul efectiv diferã de rezultatul asteptat metoda de testare poate comunica prin rezultatul sãu aceastã situatie (rezultat “boolean”) sau poate genera o exceptie. In exemplul urmãtor se generează o exceptie “run-time”:

```
public void testGet() {
    BitSet a = new BitSet();
     a.set(5);                    // setare bit 5
     if (a.get(5) == false)
        throw new FailureException();
}
```

Se poate adãuga metodei “testGet” si un test de citire a unui bit pus pe zero prin metoda “clear” a clasei “BitSet”, metodã certificatã anterior ca fiind corectã.

Utilizatorul trebuie sã scrie mai multe metode de tipul “testGet”, sã execute aceste metode si sã ia o decizie în cazul aparitiei de erori. Decizia poate fi oprirea testelor la primul test esuat sau continuarea testelor si prezentarea unui raport final cu numãrul de teste care au trecut si cu cele care nu au trecut.

Pentru a încuraja scrierea si executia de teste cât mai complete este de dorit ca aceste operatii sã necesite un efort minim din partea utilizatorilor.

Inainte de implementare sunt analizate diferite cazuri de utilizare (“use case”). De aceea, vom prezenta ceea ce ar trebui sã scrie utilizatorul care vrea sã testeze o parte din metodele clasei *BitSet* (într-o variantã posibilã):

```
     // testare metode “set” si “get” din clasa BitSet
public class BSTest extends TestCase {
    BitSet a = new BitSet();              // un obiect ptr exersare metode clasa BitSet
    public static Test suite() {                  // creare suita de teste
        return new TestSuite(BSTest.class);   // extrage metode de test din cl. BSTest
    }
      // metode de test specifice clasei verificate
    public void testSet() {               // metoda de test a metodei “set”
         a.set (3); a.set(1);
         assertEquals(a.toString().trim(),"{1, 3}");
    }
    public void testGet() {               // metoda de test a metodei “get”
         a.set(5);
         assertTrue (a.get(5));
    }
        // rulare teste si afisare rezultate
    public static void main (String[] args) {
         Test test = suite();                          // creare secventa de teste
         TestResult res = new TestResult();      // obiect pentru colectare rezultate
         test.run(res);                                // executie teste din secventa
         printResult (res);                            // extragere si afisare rezultate
    }
}
```

Clasele TestSuite, Test, TestResult sunt predefinite, ca si metodele “assertTrue”, “assertEquals” si “run”. Metoda “printResult” ar trebui scrisã tot de utilizator prin apelarea unor metode din clasa “TestResult”, pentru obtinere de rezultate. Operatiile din “main” sunt mereu aceleasi (creare secventa de teste, rulare teste, afisare rezultate) si pot fi înlocuite printr-un singur apel, pentru o afisare standard a rezultatelor.

Proiectarea bibliotecii de clase

In programarea orientatã pe obiecte simplificarea scrierii unor aplicatii (sau pãrti de aplicatii) se face prin crearea unei infrastructuri de clase si interfete (“framework”) la care utilizatorul adaugã metode si clase (derivate) specifice cerintelor sale.

Un framework oferã de obicei o solutie standard de utilizare dar si posibilitatea de a crea alte solutii, cu un efort de programare suplimentar. In cazul testelor aceste solutii se referã în principal la modul de utilizare si de afisare a rezultatelor testelor.

Biblioteca JUnit este o astfel de infrastructurã, care contine interfete, clase abstracte si câteva clase instantiabile care faciliteazã scrierea testelor si permite diverse optiuni din partea utilizatorilor.

Numãrul metodelor de test poate fi foarte mare, deoarece pentru fiecare metodã testatã ar fi necesarã cel putin o metodã de test. De aceea nu ar fi bine ca sã avem câte o clasã separatã (“top-level”) pentru fiecare metodã de test (printre altele ar putea apare si conflicte de nume între aceste clase).

Pe de altã parte, executia testelor ar trebui repetatã dupã orice modificare în clasele verificate, într-o secventã de înlãntuire automatã a testelor. Aceastã înlãntuire de teste se poate realiza prin introducerea într-o colectie Java a obiectelor ce contin metodele de test (de cãtre utilizator) si prin parcurgerea colectiei cu apelarea metodelor de test. Pentru unificarea apelãrii unor metode diverse ar trebui ca toate obiectele test sã continã o metodã comunã “run”, apelatã de obiectul care activeazã testele (numit si “TestRunner” în JUnit).

Pentru colectarea rezultatelor testelor trebuie definitã o clasã cu câteva variabile si vectori în care se memoreazã numãrul de teste trecute, numãrul de teste esuate , rezultatele testelor individuale s.a. Un obiect din aceastã clasã (numitã “TestResult” în JUnit) este transmis ca argument la metodele “run”, sau aceste metode ar putea avea ca rezultat un obiect “TestResult”.

Executia testelor poate dura destul de mult si de aceea s-a adãugat si posibilitatea vizualizãrii continue a evolutiei lor, deci o monitorizare a progresãrii testelor, în mod text si/sau în mod grafic (în afara statisticilor finale).

Variante de implementare si utilizare JUnit

Metodele de test ar putea semnala o eroare fie prin rezultatul lor, fie prin exceptii.

Alegerea între rezultat boolean sau generare de exceptie pentru metodele de test este decisã de urmãtoarea observatie: în afara erorilor asteptate (ca diferentã între un rezultat asteptat si rezultatul executiei efective), mai pot apare si exceptii neprevãzute la rularea testelor. Exemplu urmãtor produce o astfel de exceptie:

```
public void testGet() {
    a.set(-5);                  // IndexOutOfBoundsException  !!!
    if (a.get(5) == false)
        throw new FailureException();
}
```

Pentru simplificarea generãrii de exceptii în metodele de test pachetul JUnit include mai multe metode “assertXXX”,deoarece la data creãrii lui JUnit nu exista încã în Java constructia “assert”. De exemplu, metoda “assertTrue” aratã astfel:

```
protected void assertTrue (boolean cond) {      // cond = o conditie
    if (cond)
      throw new AssertionFailedException();
}
```

Utilizatorii scriu functii de test si apoi transmit adresele acestor functii unei metode care le apeleazã si colecteazã rezultatele. In Java orice functie trebuie sã facã parte dintr-o clasã.

Colectia de metode de test poate fi creatã în Java în douã moduri diferite:
- Prin crearea de obiecte pentru fiecare metodã de test si adãugarea acestor obiecte (de fapt a adreselor obiectelor) la o colectie, cum ar fi un vector. Metodele de test pot avea orice nume în acest caz, dar utilizatorul trebuie sã scrie mai mult.
- Prin trasmiterea unui obiect de tip “Class” din care, prin reflectie, sã se determine numele metodelor de test. In acest caz metodele de test trebuie sã fie publice si sã poatã fi recunoscute fatã de alte metode din clasa respectivã. Conventia JUnit este ca metodele de test (si numai ele) sã aibã un nume care începe prin sirul “test”.

Pentru concretizare vom ilustra cele douã variante în cazul a douã metode de test pentru clasa “BitSet”. Metodele numite “testGet” si “testSet” fac parte dintr-o clasã “BitTest”. Obiectul colectie folosit este de tipul “TestSuite” si contine un obiect din clasa “Vector”. Metoda “addTest” din “TestSuite” apeleazã metoda “addElement” a clasei “Vector”.

In prima variantã se creeazã obiecte din subclase interne anonime definite ad-hoc din clasa “BitTest” pentru a extrage fiecare metodã într-un obiect separat.

```
  // suita de teste creata explicit
public static Test suite() {
  TestSuite suite= new TestSuite();
  suite.addTest( new BitTest() {
     protected void runTest() { testSet(); }
  } );
  suite.addTest( new BitTest() {
     protected void runTest() { testGet(); }
  } );
  return suite;
}
```

In a doua variantã se foloseste un alt constructor din clasa “TestSuite”, care primeste un obiect de tip “Class” si extrage metodele clasei care încep cu “test”:

```
 //  suita creata prin determinare metode clasa “BitTest” la executie
public static Test suite() {
   return new TestSuite(BitTest.class);
}
```

Vectorul de teste din clasa TestSuite este folosit prin intermediul unui enumerator creat de functia “tests” (care apeleazã metoda “elements” din clasa *Vector*):

```
    // executa repetat fiecare test din vector
public void run(TestResult result) {          // din clasa “TestSuite”
     for (Enumeration e= tests(); e.hasMoreElements(); ) {
          Test test= (Test)e.nextElement();
          runTest(test, result);
     }
}
    // executa un singur test
public void runTest(Test test, TestResult result) {
```

```
    test.run(result);
}
```

De observat cã toate obiectele tester trebuie sã continã o metodã “run”, deci ar trebui ca toate clasele definite de utilizatori sã aibã o superclasã comunã, cu o metodã “run” cu argument “TestResult”. Aceastã superclasã se numeste “TestCase” în JUnit.

Clasa TestCase

Metodele de test (ca “testGet”) trebuie definite într-o clasã, dar obiecte din aceastã clasã trebuie introduse într-o colectie (vector), deci ar trebui sã aibã toate un acelasi tip si câteva metode comune. Acest (supra) tip comun este definit de clasa abstractã “TestCase”, care contine si o metodã “runTest”, care apeleazã metoda de test scrisã de utilizator (care poate avea orice alt nume).

Utilizatorul trebuie sã defineascã o subclasã a clasei “TestCase”. Exemplu:

```
public class BitTest extends TestCase {
    BitSet a,b;
    . . .
}
```

Fiecare metodã de test creeazã unul sau câteva obiecte din clasa testatã si apeleazã metode ale acestor obiecte. Obiectele pot fi create în fiecare metodã de test, astfel:

```
public void testGet() {              // verificarea metodei “get”
    BitSet a = new BitSet();         // creare obiect de tip BitSet
    a.set(5); assertTrue (a.get(5));
}
```

Pentru a evita repetarea unor instructiuni în fiecare metodã de test, obiectele necesare sunt create într-o singurã functie, numitã “setUp” (din clasa “TestCase”):

```
    protected void setUp() {
        a= new BitSet();
    }
```

O solutie ar putea fi si crearea obiectelor la încãrcarea clasei, dar atunci ar putea apare interferente între teste; functia “setUp” este apelatã implicit înainte de apelul fiecãrei metode de test (scrisã de utilizator). Este prevãzutã si o metodã “tearDown” pentru “curãtenie” dupã teste, dar ea este definitã mai rar ca “setUp” (de exemplu, pentru închidere unei conexiuni de retea folosite la teste); ea este apelatã automat dupã fiecare test.

Pentru folosirea în comun a unor variabile din clasa testatã (*BitSet*, de exemplu) se recomandã gruparea testelor unei clase într-o singurã clasã derivatã din “TestCase”

Clasa “TestCase” corespunde unui singur caz de test si este declaratã abstractã pentru a nu fi instantiatã direct (se folosesc numai obiecte din subclasele sale).

```
public abstract class TestCase extends Assert implements Test {
```

```
    private String fName;       // un nume pentru test
    public TestCase() { fName= null; }
    public TestCase(String name) {  fName= name;  }
    public int countTestCases() { return 1;  }
    protected TestResult createResult() {
       return new TestResult();
    }
    public TestResult run() {
         TestResult result= createResult();
         run(result);
         return result;
    }
    public void run(TestResult result) {
         result.run(this);
    }
    public void runBare() throws Throwable {
         setUp();
         try { runTest();}
         finally {tearDown(); }
    }
    protected void setUp() throws Exception { }
    protected void tearDown() throws Exception { }
    public String toString() {
       return getName() + "(" + getClass().getName() + ")";
    }
    //… alte metode
}
```

Extinderea clasei “Assert” permite utilizarea mai simplã a metodelor statice din aceastã clasã de cãtre utilizatori, ca si cum ar fi metode ale clasei “TestCase”.

Clasa TestSuite

O subclasã derivatã din “TestCase” reuneste de obicei testele unei clase, dar o aplicatie poate contine mai multe clase, iar rularea tuturor testelor asociate aplicatiei ar trebui sã se facã înlãntuit (automat, fãrã interventia operatorului). Clasa “TestSuite” permite combinarea mai multor teste (derivate din “TestCase” sau din “TestSuite”) într-o singurã suitã de teste. Un obiect “TestSuite” poate fi privit ca un arbore, compus din noduri (obiecte “TestCase”) si subarbori (obiecte “TestSuite”).

Pentru a putea reuni obiecte de tipurile “TestCase” si “TestSuite” într-un singur obiect “TestSuite” este necesar ca cele douã clase sã aibã un supertip comun, care este interfata “Test”. Interfata “Test” impune douã metode claselor TestCase si TestSuite:

```
public interface Test {
    public abstract int countTestCases();
    public abstract void run(TestResult result);
}
```

Clasa “TestSuite” contine un obiect *Vector*, constructori pentru completarea acestui vector si metode pentru folosirea vectorului de teste.

```
public class TestSuite implements Test {
    private Vector fTests= new Vector(10);
    private String fName;                    // un nume ptr suita de teste
    . . .
    public void addTest(Test test) {       // adauga un obiect Test la vector
        fTests.addElement(test);
    }
    public void addTestSuite(Class testClass) {    // adaugã o suitã de teste
        addTest(new TestSuite(testClass));          // cu obiecte extrase dintr-o clasa
    }
    public void run(TestResult result) {           // executare teste din vector
        for (Enumeration e= tests(); e.hasMoreElements(); ) {
            Test test= (Test)e.nextElement();
            runTest(test, result);
        }
    }
    public void runTest(Test test, TestResult result) {
        test.run(result);                    // apel metoda run din obiect Test
    }
    public Enumeration tests() {
        return fTests.elements();
    }
}
```

Mai interesant este constructorul clasei “TestSuite” care foloseste reflectia pentru extragerea metodelor publice al cãror nume începe cu “test” dintr-o clasã datã si crearea de obiecte de tip “TestCase” pentru fiecare dintre aceste metode. Utilizarea metodelor de reflectie necesitã multe verificări si poate produce diverse exceptii, cum ar fi inexistenta unor metode publice “testXXX” în clasa analizatã. De aceea vom da aici o versiune mult simplificatã a acestui constructor:

```
 public TestSuite(final Class c) { // extrage metode “testXXX” din clasa c
      fName= c.getName();
      if ( c.getSuperclass() != TestCase.class) return ;
      Method[] methods= c.getDeclaredMethods();
      for (int i= 0; i < methods.length; i++)    // adauga obiecte cu aceste metode
        addTestMethod(methods[i], c);
}
private void addTestMethod(Method m, Class c) throws Exception {
    String name= m.getName();      // nume metoda
     if (isTestMethod(m))
        addTest(createTest( c, name));
}
  // creare obiect de tip “Test” cu numele metodei
static public Test createTest(Class c, String name) throws Exception {
   Class[] args= { String.class };
```

```
   Constructor constructor = c.getConstructor(args);
   Object test = constructor.newInstance(new Object[]{name});
   return (Test) test;
}
```

Alte clase din infrastructura JUnit

Pachetul de clase JUnit contine câteva subpachete: framework, runner, textui s.a. Vom analiza pe scurt clasele din pachetul “framework”.

Clasa “Assert” reuneste mai multe metode statice care verificã o conditie si aruncã o exceptie în caz de conditie falsã. Urmeazã un fragment din aceastã clasã:

```
package junit.framework;
public class Assert {
    protected Assert() {}    // nu poate fi instantiata
    static public void assertTrue(String message, boolean condition) {
        if (!condition)
            fail(message);
    }
    static public void assertTrue(boolean condition) {
        assertTrue(null, condition);
    }
    static public void fail(String message) {
        throw new AssertionFailedError(message);
    }
    static public void assertEquals(String message, String expected, String actual) {
        if (expected == null && actual == null)
            return;
        if (expected != null && expected.equals(actual))
            return;
        throw new ComparisonFailure(message, expected, actual);
    }
    static String format(String message, Object expected, Object actual) {
        String formatted= "";
        if (message != null)
            formatted= message+" ";
        return formatted+"expected:<"+expected+"> but was:<"+actual+">";
    }
. . .    // alte metode
}
```

Pachetul “framework” contine si clasele pentru obiecte exceptie: AssertionFailedError si ComparisonError. Fragmente din aceste clase:

```
public class AssertionFailedError extends Error {
    public AssertionFailedError () {   }
    public AssertionFailedError (String message) {
        super (message);
    }
```

```
}
public class ComparisonFailure extends AssertionFailedError {
    private String fExpected;
    private String fActual;
    public ComparisonFailure (String message, String expected, String actual) {
        super (message);
        fExpected= expected;       fActual= actual;
    }
    public String getMessage() {
        if (fExpected == null || fActual == null)
            return Assert.format(super.getMessage(), fExpected, fActual);
    // … alte metode
}
```

Folosirea rezultatelor testelor

La apelarea metodei “run” din clasa ”TestSuite” se poate transmite un argument de tipul “TestResult”. Exemplu:

```
    public static void main (String[] args) {       // din clasa BitTest
        TestSuite test = suite();
        TestResult result = new TestResult();
        test.run(result);
        …   // utilizare obiect “result”
    }
```

Utilizarea obiectului cu rezultatele testelor necesitã cunoasterea metodelor clasei TestResult”. Exemplu:

```
        System.out.println (result.runCount()+ " tests");
        System.out.println (result.failureCount() + " failures");
        System.out.println (result.errorCount() + " errors");
```

Pentru simplificarea utilizării rezultatelor testelor sunt prevãzute câteva clase care afiseazã statistici si timpul necesar rulării testelor, fie în mod text, fie în mod grafic (cu o barã de progres care vizualizează evolutia unor teste de duratã). Aceste clase se numesc toate “TestRunner” si contin metoda “run” dar sunt în pachete diferite. Pentru afisare în mod text functia “main” poate contine numai instructiunea urmãtoare:

```
        junit.textui.TestRunner.run(suite());
```

Clasa “TestResult” are ca principalã actiune metoda “run” care executã un test si trateazã exceptiile ce pot apare la executia testului respectiv; “tratare” înseamnã colectarea de statistici si notificarea unor obiecte “observator”, înregistrate anterior.
Pentru a usura întelegerea am rescris metoda “run” din “TestResult” astfel:

```
    protected void run(final TestCase test) {
        startTest(test);        // anumta observatori de incepere test
        try {
```

```
            test.runBare();          // executa setUp, metoda de test si tearDown
        }
        catch (AssertionFailedError e) {addFailure(test, e); }
        catch (Throwable e) {  addError(test, e); }
        endTest (test);
    }
```

Metodele “addError” si “addFailure” realizează aceleasi operatii - adaugă un obiect eroare la un vector si notifică toti ascultătorii (observatorii) de tip “TestListener”:

```
    public synchronized void addError(Test test, Throwable t) {
        fErrors.addElement(new TestFailure(test, t));
        for (Enumeration e= cloneListeners().elements(); e.hasMoreElements(); ) {
            ((TestListener)e.nextElement()).addError(test, t);
    }
```

Ca orice clasă pentru obiecte observate, clasa “TestResult” contine un vector de obiecte ascultător si metode pentru adăugarea si stergerea de obiecte ascultător. Metodele “startTest” si “endTest” notifică si ele ascultătorii acestei clase.

Pentru modul text este implementată o clasă ascultător, numită “ResultPrinter”, iar clasa “TestRunner” contine o variabilă “ResultPrinter”. Efectul metodelor din clasa ascultător, care reactionează la notificări, este foarte simplu : afisează litera ‘F’ pentru erori “AssertionFailed”, litera ‘E’ pentru exceptii. Exemplu:

```
    public void addFailure(Test test, AssertionFailedError t) {
        getWriter().print("F");
    }
```

La evenimente produse de metoda “startTest” se afisează un punct (test trecut cu bine), iar la evenimente produse de “endTest” nu se afisează nimic.

```
public class TestRunner extends BaseTestRunner {
    private ResultPrinter fPrinter;
    public TestResult doRun(Test suite) {
        TestResult result= createTestResult();
        result.addListener(fPrinter);
        long startTime= System.currentTimeMillis();
        suite.run(result);
        long endTime= System.currentTimeMillis();
        long runTime= endTime-startTime;
        fPrinter.print(result, runTime);
        return result;
    }
```

Anexa B. Dezvoltarea de aplicatii Java

Dezvoltarea unei aplicatii Java înseamnã mai mult decât scrierea de cod, iar alegerea modului de operare si a instrumentelor de dezvoltare poate influenta mult timpul de realizare a unei aplicatii mari.

Comenzi de compilare si executie

Compilatorul si interpretorul de Java (furnizate de firma Sun Microsystem) sunt destinate a fi folosite în modul linie de comandã, deci apelate prin comenzi introduse în modul text sau într-o fereastrã consolã în modul grafic. Mediile integrate Java folosesc tot aceste programe, dar apelarea lor de cãtre om este indirectã, prin optiuni ale meniului din fereastra principalã. Exemplu de comenzi pentru rularea unui program simplu, format dintr-o singurã clasã, numitã “Test”, care contine pe “main”:

```
javac test.java
java Test
```

Existã câteva particularitãti ale dezvoltãrii de programe Java fatã de programele C.

Se recomandã ca un fisier sursã “java” sã continã o singurã clasã; dacã este o clasã publicã atunci aceastã cerintã este obligatorie.

La compilare, în comanda “javac” pot apare unul sau mai multe nume de fisiere sursã (cu extensia “java” mentionatã explicit). Dacã fiecare fisier contine o clasã si are acelasi nume cu clasa, atunci este suficient ca în comanda de compilare sã se dea doar numele fisierului ce contine functia “main” (începând cu care se fac referiri la toate celelalte clase folosite în aplicatie). Compilatorul Java poate cãuta automat clasele referite în alte fisiere sursã, dar cãutarea se face dupã numele de fisier (clasã). De exemplu, putem avea 3 fisiere sursã numite Nod.java, SList.java si TestList.java, fiecare contine o clasã cu acelasi nume. Comanda de compilare poate fi:

```
javac TestList.java
```

Compilatorul Java poate folosi si fisiere de tip “class” sau “jar” pentru rezolvarea referintelor la alte clase. Aceste fisiere se pot afla în acelasi director sau în alte directoare, mentionate în optiunea de compilare “-classpath”. Dupã aceastã optiune poate urma o secventã de directoare (o cale) si un nume de fisier “jar”.

Calea (sau cãile) pe care sunt cãutate clasele sunt stabilite fie prin comenzi sau prin componente ale sistemului de operare (Windows Explorer în Windows XP), fie sunt indicate în comanda de compilare prin optiunea “-classpath”.

De exemplu, fisierul care contine teste pentru o clasã se numeste “BitTest.java” si foloseste clase din biblioteca “junit.jar”. Pentru compilare folosim comanda:

```
javac –classpath junit.jar BitTest.java
```

sau modificãm calea cãtre clase printr-o comandã separatã:

```
set classpath=%classpath%;c:\junit\junit.jar
javac BitTest.java
```

Cea mai mare parte din clasele folosite sunt clase predefinite SDK, aflate în biblioteca “rt.jar”, pentru care ar trebui să existe o cale implicită de căutare.

Ca rezultat al compilării se produc fisiere de tip “class”, câte unul pentru fiecare clasă din fisierele compilate.

La interpretarea codului intermediar, comanda “java” trebuie să conțină numele clasei care contine functia statică “main” cu care începe executia. Numele clasei poate fi diferit de numele fisierului sursă compilat. Alte clase necesare executiei sunt căutate în fisierele “jar” mentionate sau pe căile mentionate (optiunea –classpath).

De exemplu, executia programului de test compilat anterior, care foloseste si clase din biblioteca “junit.jar” (aflată în acelasi director) se va face astfel:

```
java –classpath .;junit.jar BitTest
```

Dacă biblioteca junit.jar se află într-un alt director decât fisierul BitTest.class:

```
java –classpath .;c:\junit\junit.jar BitTest
```

Fisierele de tip “jar” sunt arhive compuse din fisiere “class” (posibil si alte tipuri de fisiere) si corespund fie unei biblioteci de uz mai general, fie concentrează într-un singur fisier toate clasele unei aplicatii (este forma de distribuire a aplicatiilor).

Pentru executia unei aplicatii ale cărei clase sunt toate arhivate vom scrie, de ex.:

```
java –jar SwingDemo.jar
```

Un fisier “jar” poate să aibă o structură de directoare, care să reflecte ierarhia de pachete de clase din compunerea arhivei (nu este doar o secventă liniară de fisiere).O arhivă “jar” poate fi creată si prelucrată cu ajutorul programului “jar.exe”.

Clasele care contin o instructiune “package p” trebuie introduse intr-un director cu numele “p”, iar comenzile de compilare si executie se dau din afara directorului “p”. Fie clasele urmatoare :

```
  // fisier a.java
package teste;
import teste.*;
public class a {
  public static void main (String arg[]) {
    b.print ("a->b");           // sau  teste.b.print (“a->b”);  // fara “import”
  }
}
  // fisier b.java
package teste;
public class b {
  public static void print (String s) {
    System.out.println(s);
  }
```

}

Fisierele “a.java” si “b.java” se introduc într-un subdirector cu numele “teste” iar comenzile urmãtoare se dau din directorul pãrinte al directorului “teste” :

```
javac teste/a.java
java teste.a
```

Fisiere de comenzi

Pentru pregãtirea unei aplicatii mai mari în vederea distributiei si pentru refacerea arhivei livrabile dupã modificãri pot fi necesare mai multe comenzi, cu eventuale optiuni. Aceste comenzi sunt reunite într-un fisier de comenzi, pentru a simplifica procedurile de operare cerute de modificarea aplicatiei.

Executia unor comenzi poate fi conditionatã de reusita unor comenzi anterioare si deci existã anumite dependente între actiunile (sarcinile) destinate rulãrii aplicatiei. Exprimarea acestor dependente presupune anumite conventii si poate apela sau nu la facilitãti oferite de sistemul de operare gazdã pentru programare la nivel de “shell” (fiecare sistem de operare, dar mai ales sisteme de tip Unix/Linux au unul sau câteva limbaje de comenzi, pentru exprimarea acestor dependente).

Interpretarea fisierului de comenzi poate fi fãcutã de un program independent de sistemul de operare, numit de obicei “make” (“gnumake”, “nmake”, s.a.). Fisierul de comenzi interpretat de “make” poate avea un nume fix (makefile, de exemplu) si este un fisier text care respectã anumite reguli.

Programul “ant” (Apache) este un fel de “make” fãrã dezavantajele lui “make” si independent de sistemul de operare gazdã (este scris în Java). Fisierele prelucrate de “ant” sunt fisiere XML care descriu o serie de obiective (targets) ce pot depinde unele de altele. Fãrã a intra în amãnunte de utilizare a programului “ant” prezentãm un exemplu de fisier “build” din documentatia “ant”:

```
<project name="MyProject" default="dist" basedir=".">
    <description>  simple example build file   </description>
  <!-- set global properties for this build -->
  <property name="src" location="src"/>
  <property name="build" location="build"/>
  <property name="dist"  location="dist"/>

  <target name="init">
    <!-- Create the time stamp -->
    <tstamp/>
    <!-- Create the build directory structure used by compile -->
    <mkdir dir="${build}"/>
  </target>

  <target name="compile" depends="init"
       description="compile the source " >
```

```
    <!-- Compile the java code from ${src} into ${build} -->
    <javac srcdir="${src}" destdir="${build}"/>
  </target>

  <target name="dist" depends="compile"
        description="generate the distribution" >
    <!-- Create the distribution directory -->
    <mkdir dir="${dist}/lib"/>

    <!-- Put everything in ${build} into the MyProject-${DSTAMP}.jar file -->
    <jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar" basedir="${build}"/>
  </target>

  <target name="clean"
        description="clean up" >
    <!-- Delete the ${build} and ${dist} directory trees -->
    <delete dir="${build}"/>
    <delete dir="${dist}"/>
  </target>
</project>
```

Medii integrate de dezvoltare

Un mediu integrat Java (IDE= Integrated Development Environment) integreazã într-un singur produs toate programele necesare dezvoltării de aplicatii: un editor de texte, compilator, interpretor, depanator, program de vizualizare a documentatiei, s.a. Mai corect, un IDE permite utilizarea unitarã, prin meniuri de optiuni, a acestor programe. In cazul limbajului Java, aproape toate mediile folosesc tot programele din Java SDK (de la firma “Sun”) – javac, java, jar etc., dar apelate indirect din IDE.

Mediile integrate pot fi foarte diferite ca facilitãti si ca resurse solicitate; mediul JCreator (Xinox) are cca 2 Moct si poate fi folosit pe orice calculator cu Windows-95 în timp ce mediul NetBeans ocupã peste 30 Moct pe disc si poate fi folosit eficient pe un sistem cu Windows-XP. Aceste douã produse sunt gratuite si deosebit de utile.

Un mediu integrat oferã avantaje multiple pentru dezvoltarea de programe:

- Localizare în textul sursã pe liniile cu erori;
- Ajutor (eventual dependent de context) pentru numele claselor si metodelor, lucru deosebit de important în conditiile cresterii continue a numãrului de clase si metode;
- Facilitãti pentru specificare cãi de cãutare si biblioteci folosite;
- Creare automata subdirectoare pentru pachete de clase (clase ce contin “package”);
- Facilitãti pentru depanare care folosesc interfata graficã;
- Colorarea diferitã a unitãtilor sintactice (cuvinte cheie, comentarii, s.a.);
- Crearea si utilizarea de directoare pentru toate fisierele unei aplicatii.
- Generarea unui schelet de program, în functie de tipul aplicatiei;
- Autocompletare: completare automatã nume lungi pe baza unui prefix introdus de utilizator;

- Crearea automatã de proiecte, eventual de fisiere pentru “ant”;

Medii vizuale

Un mediu vizual este un mediu integrat care permite si generarea de cod pentru crearea si interactiunea componentelor unei interfete grafice; utilizatorul “vede” aceste componente pe ecran, poate modifica proprietãtile si legãturile dintre ele.

Programarea unei interfete grafice necesitã multe linii de cod, dar este o muncã de rutinã care urmeazã anumite tipare. In plus, este necesarã o perioadã de experimentare pentru ajustarea dimensiunilor si pozitiei componentelor vizuale în fereastra principalã a aplicatiei.

O solutie este generarea automatã a codului sursã necesar pentru crearea si interactiunea componentelor unei interfete grafice, cu reducerea sau chiar eliminarea codului scris manual. Proiectantul interfetei are la dispozitie o paletã de componente precum si posibilitatea configurãrii si conectãrii acestor componente fãrã a scrie cod si cu afisarea continuã pe ecran a rezultatului deciziilor sale.

Programul care permite acest mod de lucru (si foloseste la rândul lui o interfatã graficã cu utilizatorul) se numeste fie mediu vizual, fie constructor de aplicatii, fie asamblor de componente, fie mediu pentru dezvoltarea rapidã de aplicatii (RAD). Un astfel de mediu permite obtinerea rapidã a unui prototip al interfetei grafice sau chiar al întregii aplicatii, prototip care poate fi prezentat beneficiarului si apoi îmbunãtãtit atât ca aspect si usurintã de folosire, cât si ca performante.

O componentã software este un modul de program care realizeazã complet o anumitã functie ("self-contained") si care poate fi usor cuplat cu alte componente. Crearea unei aplicatii prin asamblarea de componente software este mai putin flexibilã decât programarea manualã a unei aplicatii, în sensul cã nu se pot defini clase derivate si nu se pot stabili orice relatii între clasele existente (legãturile dintre clase sunt generate si ele automat dupã o schemã prestabilitã).

Clasele componentelor software nu sunt disponibile de obicei în sursã, iar crearea de obiecte cu anumite proprietãti nu se face prin programare. De exemplu, pentru crearea unui obiect *JLabel* (o etichetã cu text asociatã altei componente), vom selecta clasa (componenta) *JLabel* dintr-un set de componente disponibile, vom aduce eticheta în pozitia doritã (prin “târâre” pe ecran) si vom stabili continutul si aspectul textului prin modificãri interactive ale proprietãtilor afisate de mediu.

Pentru specificarea interactiunilor dintre componente (apeluri de metode sau rãspuns la evenimente generate de componente) poate fi necesarã si scrierea unor secvente de cod (intercalate în codul generat automat de mediu), pe lângã actiunile de cuplare” a obiectelor si de modificare a proprietãtilor.

Nu orice clasã trebuie transformatã într-o componentã standard si nu orice aplicatie se obtine mai usor si mai rapid prin asamblarea de componente.

Numãrul de componente manipulate de mediu nu este limitat, dar pentru ca programul asamblor de componente sã poatã recunoaste si folosi aceste componente ele trebuie sã respecte anumite conventii. Prin “model de componente” se înteleg aceste conventii pe care trebuie sã le respecte clasele (vizuale si nonvizuale) ale cãror obiecte sã poatã fi manipulate printr-un program. Mediul trebuie sã “descopere”

metodele clasei, tipurile de evenimente generate si “proprietãtile” obiectelor, informatii cunoscute si folosite de programator în scrierea manualã a programelor.

Granularitatea si complexitatea componentelor standard poate fi foarte diferitã.

Respectarea modelului de componente permite asamblarea de componente provenind din diverse surse (de la diversi furnizori), precum si folosirea lor în diferite medii vizuale pentru construirea de aplicatii. Componentele cu acelasi rol ar trebui sã fie interschimbabile si manipulabile în acelasi fel de cãtre un mediu cu componente.

Modelul de componente standard Java se numeste JavaBeans, iar o componentã este numitã “bean” (un “bob” sau o granulã). Multe clase Swing (JFC) sunt componente standard si pot fi folosite direct într-un mediu vizual. Pentru definirea de componente standard trebuie cunoscute si respectate cerintele modelului JavaBeans.

Componentele JavaBeans poate fi folosite pentru crearea altor componente Beans compuse, pentru crearea de apleti sau de aplicatii, fãrã a scrie cod sau cu un minim de cod sursã scris de programator.

Cerintele JavaBeans pot fi rezumate astfel:

- Clasele au numai constructori fãrã argumente, pentru a simplifica instantierea lor automatã; variabilele clasei au de obicei valori implicite si pot fi modificate ulterior prin metode “setProp”, unde “Prop” este numele variabilei (proprietãtii).
- Clasele trebuie sã continã metode publice pentru obtinerea si (eventual) modificarea unor atribute ale obiectelor (numite si “proprietãti”), atribute de interes pentru mediul vizual si pentru alte componente. Numele acestor metode trebuie sã respecte un anumit “tipar” si contin numele proprietãtii. De exemplu, cea mai importantã proprietate a unei etichete (obiect de tip *JLabel*) este textul afisat pe ecran ca etichetã. Variabila “private” de tip *String* din clasa *JLabel* este proprietatea numitã “text” si este accesibilã prin metodele urmãtoare:

```
public void setText (String text)          // citire valoare proprietate
public String getText ( )                  // modificare valoare proprietate
```

Componentele pot contine si alte metode publice, preferabil fãrã argumente.

Ordinea de apelare a metodelor de tip "set" si "get" nu ar trebui sã conteze pentru cã proprietãtile ar trebui sã fie independente unele de altele. Metodele "set" trebuie sã verifice argumentele si sã genereze exceptii la erori în datele primite.

Existenta metodelor publice de modificare a proprietãtilor permite definirea de constructori fãrã argumente, mai usor de utilizat de cãtre mediul cu componente (în programarea "manualã" se obisnuieste ca atributele obiectelor sã fie stabilite la construirea lor, pentru cã este mai comod asa (nu se mai apeleazã alte metode).

- Clasele pot contine si proprietãti “legate” (“Bounded Properties”), a cãror modificare este automat semnalatã claselor interesate si care au fost înregistrate ca “ascultãtori” la modificarea acestor proprietãti. Clasa *JLabel* are 5 proprietãti legate proprii pe lângã cele mostenite de la superclasa sa: imaginea afisatã ca etichetã (“icon”), alinierea orizontalã a textului, alinierea verticalã s.a. Modificarea unei proprietãti legate generează un fel de eveniment:

```
public void setHorizontalTextPosition(int textPosition) {
    int old = horizontalTextPosition;
```

```
        firePropertyChange ("horizontalTextPosition", old, horizontalTextPosition);
        repaint();
 }
```

Fiecare clasã cu proprietãti legate contine (sau mosteneste) o listã de ascultãtori:

```
private PropertyChangeSupport list = new PropertyChangeSupport (this);
```

Adãugarea sau eliminarea de ascultãtori la listã se face cu metodele urmãtoare:

```
public void addPropertyChangeListener ( PropertyChangeListener pcl) {
   list. addPropertyChangeListener (pcl);
}
public void removePropertyChangeListener ( PropertyChangeListener pcl) {
   list. removePropertyChangeListener (pcl);
}
```

Clasa interesatã de modificarea unei proprietãti dintr-o altã clasã trebuie sã continã o metodã cu antetul urmãtor:

```
   public void propertyChange (PropertyChangeEvent e);
```

- Componentele standard pot genera evenimente, fie de tipurile predefinite, fie de alte tipuri (subtipuri ale clasei *EventObject*). Comunicarea prin evenimente se impune pentru clasele care semnaleazã actiuni ale operatorului uman, dar este utilã si pentru alte clase deoarece foloseste un mecanism standard de comunicare între clase (standardizeazã numele si argumentele metodelor apelate).
- O clasã compatibilã Java Beans are în general atasatã o clasã descriptor (care respectã interfata *BeanInfo*), cu informatii despre clasa “bean”. Numele clasei descriptor este derivat din numele clasei descrise, urmat de “BeanInfo”. Aceastã metodã de “introspectie” se adaugã mecanismului de “reflexie” utilizabil pentru orice clasã Java, pentru determinarea la executie a numelor clasei, metodelor, variabilelor.
- Orice clasã compatibilã Java Beans trebuie sã implementeze interfata *Serializable* pentru a putea folosi metodele de salvare si refacere a obiectelor în fisiere (metodele “writeObject” si “readObject”). Prin serializare proprietãtile modificate devin persistente si nu mai trebuie restabilite prin apeluri de metode sau folosind un editor de proprietãti, la utilizãri ulterioare ale obiectelor.
- Clasele Java Beans mai pot avea asociate un editor de proprietãti si o clasã pentru adaptarea sau individualizarea lor (“Customizer”). Prin adaptare se poate restrânge numãrul de proprietãti etalate de clasã sau se pot impune anumite restrictii de apelare a metodelor clasei.

Clasele care formeazã o componentã, împreunã cu fisierul imagine asociat componentei si cu fisierul manifest sunt reunite într-o arhivã “jar”, forma de livrare a unei componente standard.

Anexa C. Versiuni ale limbajului Java

Diferente între versiunile importante

Cele mai importante versiuni ale limbajului Java sunt:
1.0 (1995), 1.1(1996) , 1.2 (1997-98), 1.3 (2000), 1.4 (2003-04), 1.5 (2004)

Diferentele de la o versiune la alta sunt de douã feluri:
- Adãugarea de noi clase si pachete (directoare) la biblioteca de clase predefinite
- Modificãri în limbajul Java, cu pãstrarea compatibilitãtii la nivel de fisiere sursã si de fisiere "class".

Compatibilitate înseamnã cã vechile programe pot fi compilate si executate fãrã modificãri si cu noile versiuni ale programelor "javac" si "java", chiar dacã mai apar avertismente de tipul "deprecated class" (clasa învechitã, ce ar trebui înlocuitã cu o altã clasã mai nouã).

Ultimele versiuni (4 si 5) solicitã si resurse hardware mai importante (memorie, vitezã de calcul si spatiu pe disc) pentru a putea fi utilizate eficient.

Numãrul mai mare de clase la fiecare nouã versiune are mai multe explicatii:
- Reformarea unor pachete de clase existente, fie pentru crearea de noi facilitãti, fie pentru îmbunãtãtirea performantelor; asa s-a întâmplat cu clasele colectie, cu clasele pentru interfete grafice (trecerea de la AWT la JFC), cu clasele de I/E (trecrea de la pachetul "io" la pachetul " nio");
- Adãugarea unor pachete pentru aplicatii noi sau pentru o abordare diferitã a unor aplicatii acoperite anterior: clase pentru lucrul cu expresii regulate, clase pentru lucrul cu fisiere XML, clase pentru aplicatii distribuite s.a.

Pentru comparatie prezentãm dimensiunea principalei biblioteci de clase (arhiva "rt.jar" din (sub)directorul "jre/lib") în câteva versiuni:

Java 1.2	~10,5	Moct
Java 1.4	~26	Moct
Java 1.5	~35	Moct

Noutãti în Java 1.2 (1.3) fatã de Java 1.1

Modificãri în limbajul Java practic nu au mai existat de la introducerea claselor incluse (înversiunea 1.1) si pânã la versiunea 1.4.

Toatã infrastructura claselor colectie, cu interfete, clase abstracte si clase direct utilizabile a apãrut începând din versiunea 1.2. Inainte existau numai câteva clase separate: *Vector, Stack, Hashtable* si *BitSet* si interfata *Enumeration.*

Cu ocazia aparitiei noului "framework" pentru colectii s-au mai modernizat si clasa *Vector* (denumitã acum *ArrayList*) si interfata *Enumeration* , sub numele de *Iterator*.

Infrastructura claselor Swing (JFC) si noul model de tratare a evenimentelor au fost si ele incluse oficial din versiunea 1.2, desi puteau fi folosite ca bibliotecã separatã si anterior (cu versiunea 1.1 care a introdus si conceptul de clase incluse).

Noutãti în Java 1.4 fatã de Java 1.2 si 1.3

Singura noutate de limbaj este introducerea asertiunilor, într-o formã asemãnãtoare constructiei “assert” din limbajul C.

Instructiunea “assert” poate avea douã forme:

```
assert boolean_exp ;
assert boolean_exp : error_exp ;
```

In ambele forme se verificã o asertiune, adicã o expresie booleanã, presupusã a fi adevãratã si, în caz de asertiune falsã se aruncã o exceptie *AssertionError*. Dacã este adevãratã asertiunea, programul continuã ca si cum ea nu ar fi existat. Optional, se poate specifica un cod de eroare (numeric sau simbolic), transmis apoi ca argument la constructorul obiectului *AssertionError* (cea de a doua formã).

Instructiunile *assert* nu sunt luate în considerare decât dacã se folosesc anumite optiuni la compilare si la interpretarea codului, pentru cã ar putea exista în programe mai vechi functii sau variabile cu numele “assert”.

Asertiunile se folosesc pentru a facilita exprimarea unor verificãri asupra unor erori putin probabile, dar totusi posibile si care ar împiedica continuarea programului. Ele pot avea si un rol de documentare a unor conditii presupuse a fi respectate. Exemplul urmãtor foloseste o variabilã “x” care nu poate avea decât valorile 0 sau 1; în caz cã apare o altã valoare pentru x se genereazã o exceptie cu transmiterea valorii lui x:

```
if (x == 0) {
  ...
}
else {
  assert x == 1 : x;  // x must be 0 or 1.
  ...
}
```

Noutãti în Java 1.5 fatã de Java 1.4

Aici apar cele mai multe modificãri în limbaj, cauzate probabil si de aparitia limbajului C# ca un concurent direct al limbajului Java.

S-a introdus cuvântul cheie *enum* pentru tipuri definite prin enumerare de valori, ca si în limbajul C. Exemplu:

```
public enum StopLight { red, yellow, green };
```

Clasele sablon (generice) pentru colectii se definesc si se folosesc asemãnãtor cu clasele “template” din C++ si din C#. Numele clasei poate fi urmat de un nume de tip între paranteze unghiulare. Pentru clase colectie acesta este tipul elementelor colectiei. Toate clasele colectie existente (si unele noi) au si o variantã de clasã sablon.

Pentru iterare mai simplã pe clase colectie s-a introdus o formã nouã pentru instructiunea *for*. Exemplu:

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

Se pot defini si folosi functii cu numãr variabil de argumente. Exemplu:

```
void argtest(Object ... args) {
      for (int i=0;i <args.length; i++) {
      }
   }

argtest ("test", "data");
```