Cuprins curs

CLASA Object

În Java nu este permisă moștenirea multiplă. Mai mult, clasele formează o structură de arbore în care rădăcina este clasa `Object`, a cărei definiţie apare în pachetul `java.lang`. Orice clasă extinde direct (implicit sau explicit) sau indirect clasa `Object`. Astfel, orice clasă din Java moștenește metodele din clasa `Object`.

Cele mai importante dintre acestea sunt:

- `public boolean` **`equals`**`(Object obj)`

  Compară obiectul curent cu cel primit ca parametru și returnează true dacă sunt egale. În clasa Object metoda testează egalitatea referințelor la obiecte (întoarce valoarea *true* numai dacă cele două obiecte comparate sunt cu adevărat identice, adică au *aceeaşi adresă* în memorie)
  Metoda `equals` trebuie să implementeze o **relaţie de echivalenţă** (reflexivă, simetrică, tranzitivă). Este recomandabil ca la comparare unui obiect cu `null` sau cu un obiect de alt tip metoda `equals` să întoarcă `false`.
  Observaţie. Metoda `equals` este folosită de metodele de căutare în *colecţii* pentru testarea egalității.

- `public int` **`hashCode`**`()`

  Returnează codul (de dispersie) asociat obiectului curent. Pentru clasa `Object` acesta se calculează pornind de la adresa de memorie a obiectului.
  Astfel, prin metoda `hashCode` fiecărui obiect îi este asociat un număr întreg (numit cod de dispersie, valoare hash sau „hash code”). Această metodă trebuie să respecte următoarele reguli:
  - valoarea asociată de `hashCode` unui obiect nu se poate schimba dacă obiectul nu se modifică
  - metoda `hashCode` apelată pentru două obiecte egale conform metodei `equals` trebuie să întoarcă aceeaşi valoare (**două obiecte egale trebuie să aibă aceeaşi valoare hash**)
  - este recomandabil ca două obiecte diferite să aibă valori hash diferite (în acest fel putând fi îmbunătăţită performanţa tabelelor de dispersie) sau cel puţin ca valorile să fie uniform distribuite într-o plajă de valori

  Observaţie. Este necesar ca suprascrierea metodei `equals` să fie însoţită de suprascrierea metodei `hashCode`, necesitate ce se observă cel mai bine în lucrul cu dicţionare (Map)

Un posibil algoritm de generare a valorii hash a unui obiect este dat, spre exemplu, în cartea

Joshua Bloch – **Effective Java,** Second Edition, Addison-Wesley, 2008

Astfel, se porneşte calculul valorii de la o valoare întreagă nenegativă:

```
int val = 17;
```

Pentru fiecare câmp `f` care intervine în metoda `equals` se calculează o valoare hash `c` a sa şi se modifică valoarea `val` calculată astfel:

```
val = val * 31 + c
```

Valoarea hash `c` asociată câmpului `f` se poate calcula astfel:

i. dacă `f` este de tip `boolean` `c = f ? 0 : 1`
ii. dacă `f` este de tip `byte`, `char`, `short`, `int`, atunci `c = (int) f`
iii. dacă `f` este de tip `long`, atunci `c = (int) (f ^ (f >>> 32))`
iv. dacă `f` este de tip `float` , atunci `c = Float.floatToIntBits(f)`
v. dacă `f` este de tip `double`, se calculează

```
long l = Double.doubleToLongBits(f)
```

şi pentru variabila `l` se aplică iii:

```
c = (int) (l ^ (l >>> 32))
```

vi. pentru un tablou, se consideră fiecare element şi se aplică regulile anterioare recursiv
vii. dacă `f` este referinţă către un obiect, atunci

```
c = (f == null ? 0 : f.hashCode())
```

Numărul `31` a fost ales pentru ca este număr prim impar şi pentru că înmulţirea cu `31` se poate optimiza folosind shiftarea pe biţi:

```
val * 31 == (val << 5) - val
```

Se putea alege orice număr prim, chiar diferite la fiecare modificare a lui `val`.

- `public String` **`toString`**`()`

  Returnează reprezentarea obiectului curent sub formă de şir de caractere (de obiect din clasa String). În clasa Object metoda returnează un şir de forma `java.lang.Object@`<hash_code>

- `public final Class` **`getClass`**`()`

  Returnează un obiect din clasa `Class`, care conține informații despre clasa din care face parte obiectului curent.

- `protected Object` **`clone`**`()`

  Creează și returnează o copie a obiectului curent.

  Observație. Se poate folosi dacă clasa sau una dintre superclasele sale implementează interfața `Cloneable` (vom reveni asupra interfețelor)

- `protected void` **`finalize`**`()`

  Apelată de garbage collector (când obiectul nu mai este referit)

Exemplu Rescrierea metodelor `equals`, `hashCode`, `toString`

```
import java.util.*;
class Pereche{
      int x;int y;
      Pereche(){}
      Pereche(int x,int y){ this.x=x;  this.y=y; }
      public String toString(){   return "("+x+", "+y+")"; }
      public boolean equals(Object o){
            if(o==null) return false;
            if(!(o instanceof Pereche))
            //if(o.getClass()!=this.getClass())
                  return false;
            Pereche p=(Pereche)o;
            return ((x == p.x) && (y == p.y));
      }
      public int hashCode(){
            int rez=17;
            rez=rez*31+x;
            rez=rez*31+y;
            return rez;
      }
}
class ExpPereche{
  public static void main(String aa[]){
      Pereche p1=new Pereche(3,4);
      Pereche p2=new Pereche(1,2);

      System.out.println("p1="+p1.toString());
      System.out.println("p2="+p2);//se apeleaza p2.toString()

      ArrayList p=new ArrayList();
      p.add(p1);
      p.add(p2);

/*metoda indeof returneaza pozitia pe care apare o pereche, -1 daca nu
exista foloseste pentru cautare equals*/
      System.out.println("Pozitia "+p.indexOf(new Pereche(1,2)));

      HashMap hm=new HashMap();
      hm.put(p1,"primul");//cheie,valoare
      hm.put(p2,"al doilea");

//cautarea valorii pentru o cheie-foloseste valoarea hash a cheii
      System.out.println(hm.get(new Pereche(1,2)));
  }
}
```

Observații.

1. Dacă vom comenta metoda `equals` din clasa `Pereche`, atunci metoda `indexOf` va returna `-1`
2. Dacă vom comenta metoda `hashCode` din clasa `Pereche`, atunci metoda `get` va returna `null`

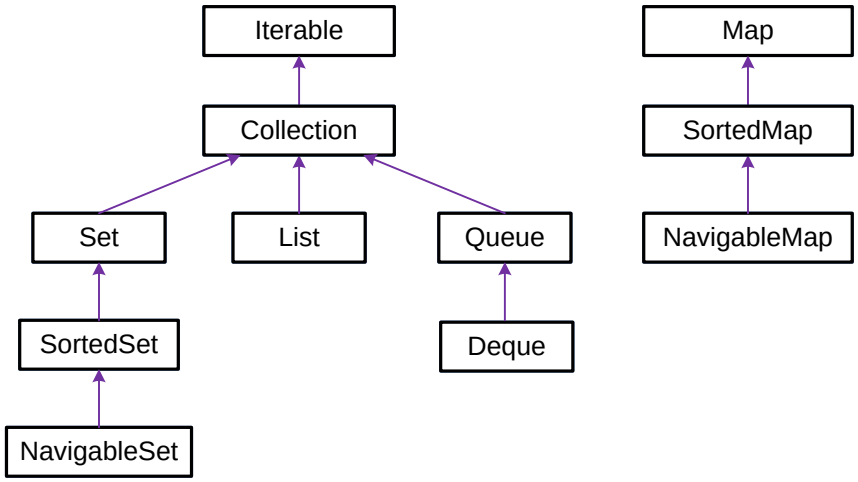
COLECȚII

O **colecţie** este un obiect care grupează mai multe obiecte într-o singură unitate, aceste obiecte fiind organizate în anumite forme.

Prin intermediul colecţiilor avem acces la diferite tipuri de date cum ar fi vectori, liste înlănţuite, stive, mulţimi matematice, tabele de dispersie, etc.

Colecţiile sunt folosite atât pentru memorarea şi manipularea datelor, cât şi pentru transmiterea unor informaţii de la o metodă la alta.

Menţionăm că ***interfeţele*** reprezintă nucleul mecanismului de lucru cu colecţii, scopul lor fiind de a permite utilizarea structurilor de date independent de modul lor de implementare (stabilesc metodele pe care trebuie să le pună la dispoziţie fiecare tip de colecţie). Vom înţelege mai bine acest mecanism când vom discuta despre interfeţe.



Majoritatea tipurilor care fac parte din categoria colecţii puse la dispoziţie de Java se află în pachetul `java.util`.

Există mai multe tipuri de colecţii, corespunzătoare interfeţelor amintite:

- **mulţime** (`HashSet`, `TreeSet`) - este o colecţie care nu conţine duplicate şi este menită să abstractizeze noţiunea matematică de mulţime
- **listă**
  - `LinkedList`
  - `ArrayList` (nesincronizat), `Vector` (sincronizat)
  - `Stack`
  - `PriorityQueue`
  - `ArrayDeque` (coadă dublă)
- **dicţionare de perechi cheie-valoare/ tabele de dispersie** (`HashMap`, `HashTable`, `IdentityHashMap`, `TreeMap`)- asociază unei *chei* o anumită *valoare*. Funcţia nu este neapărat injectivă, deci mai multor chei li se poate asocia aceeaşi valoare. Atât cheile, cât şi valorile sunt obiecte.

Alegerea unei anumite clase depinde de natura problemei ce trebuie rezolvată.

Mai multe detalii privind aceste clase (modul în care sunt implementate, performanţe…) se pot găsi în cartea **M. Naftalin, P. Wadler - Java Generics and Collections, O'Reilly, 2007**

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

- au definită metoda `toString`, care returnează o reprezentare ca şir de caractere a colecţiei respective
- permit crearea de **iteratori** pentru parcurgerea tuturor elementelor colecţiei unul câte unul (implementează interfaţa `Iterable`), cu excepţia dicţionarelor, şi de aceea pentru ele este posibilă utilizarea instrucţiunii for de tip „for each" (vezi exemplele de mai jos)
- au atât constructor fără argumente cât şi un constructor care acceptă ca argument o altă colecţie

Dar ce tip de obiecte poate conţine o colecţie?

Până la versiunea 5 nu se putea specifica tipul obiectelor introduse într-o colecţie, o colecţie de date punând conţine obiecte de orice tip (chiar de tipuri diferite, ele fiind considerate de tip `Object`). Chiar dacă obiectele conţinute în colecţie erau de acelaşi tip, nu exista o modalitate ca acesta să fie asociat colecţiei. Era în sarcina programatorului să ştie ce tipuri au obiectele introduse în colecţie, fiind necesară o **conversie explicită** la acel tip atunci când se accesează un element (tipul întors de o metodă care accesează un element al unei colecţii era `Object`). Dacă programatorul greşeşte tipul elementului, acest lucru nu se poate verifica la compilare, existând riscul apariţiei unei excepţii la execuţie de tipul `ClassCastException`.

Exemplu

```
import java.util.*;

class ExpColectii4{
      public static void main(String[] args) {

            ArrayList a=new ArrayList();
            a.add("abcd");
            a.add(new Integer(2));
            a.add("xyz");

            String s=(String)a.get(0);//trebuie cast
            System.out.println(s);

            Integer oi=(Integer)a.get(1);
            int vi=oi.intValue();
            System.out.println(vi);

            //s=(String)a.get(1);//compileaza, eroare la rulare


            System.out.print("Elementele sunt: ");
            for(int i=0;i<a.size();i++)
                System.out.print(a.get(i)+" ");//apeleaza toString

       }
}
```

Din versiunea 5 au fost introduse tipuri generice sau tipuri parametrizate. Astfel, programatorii pot specifica tipul obiectelor cu care o anumită clasă lucrează prin intermediul parametrilor de tip.

```
ArrayList<String> a=new ArrayList<String>();
ArrayList<String> a=new ArrayList<>();//din versiunea 7
```

Pentru a compila o sursa cu o versiune mai veche a JDK-ului pentru a vedea ce era permis şi ce nu în versiunile mai vechi puteţi folosi opţiunea `-source` a comenzii javac, de exemplu:

```
javac -source 1.4 ExpVector.java
```

Se poate lucra însă şi fără specificarea acestui tip, ca în versiunile anterioare

```
ArrayList a=new ArrayList();
```

Exemplu

```
import java.util.*;
class ExpColectii5{
   public static void main(String[] args) {
       ArrayList<String> a=new ArrayList<String>();
       a.add("abcd");
       //a.add(new Integer(2));//eroare la COMPILARE
       a.add("xyz");
       String s=a.get(0);//nu mai trebuie cast
       System.out.println(s);
       System.out.print("Elementele sunt: ");
       for(int i=0;i<a.size();i++)
            System.out.print(a.get(i)+" ");
       System.out.print("\nElementele afisate cu for-each: ");
       for(String v:a)
            System.out.print(v+" ");
       System.out.println();
      System.out.println("Direct "+a);// toString pentru colectie
//autoboxing/unboxing - tipuri primitive, clase infasuratoare
       ArrayList<Integer> ai=new ArrayList<Integer>();
      // ! Nu există ArrayList<int>
       ai.add(3); //autoboxing int-> Integer
       ai.add(5);
       ai.add(7);

       int x=ai.get(1); //unboxing Integer -> int
       System.out.println(x);

       System.out.print("Elementele afisate cu for-each sunt: ");
       for(int vi:ai)
            System.out.print(vi+" ");
      }
}
```

Observații.

1. Precizând tipul obiectelor care vor fi introduse în colecţie nu mai este necesară conversia explicită la extragerea unui element.
2. În cazul folosirii tipurilor generice, încercarea de a utiliza în cadrul unei colecţii a unui element necorespunzător ca tip va produce o **eroare la compilare**, spre deosebire de varianta anterioară ce permitea doar aruncarea unei excepţii la execuţie de tipul `ClassCastException` în cazul folosirii incorecte a tipurilor
3. Faptul că anumite clase sunt parametrizate este utilizat **doar de compilatorul Java** nu şi de maşina virtuală, codul generat de compilatorul Java putând fi executat şi de versiunea 1.4 a platformei.

• Liste. Clasa ArrayList

`ArrayList` şi `LinkedList` sunt două implementări ale unei liste în Java. Alegerea colecției folosite într-un program depinde de natura problemei ce trebuie rezolvată.

	get	add	contains	next	remove
ArrayList	**1**	**1**	**n**	**1**	**n**
LinkedList	**n**	**1**	**n**	**1**	**1**

Accesarea unui element este mai rapidă (timp constant) pentru `ArrayList`, în timp ce pentru `LinkedList` este lentă, accesarea unui element într-o listă înlănțuită necesitând parcurgerea secvențială a listei pâna la elementul respectiv.
La eliminarea unui elemente din listă, pentru `ArrayList` este necesar un proces de reindexare (shift la stânga) a elementelor, în timp ce pentru `LinkedList` este rapidă, presupunând doar simpla schimbare a unor legături.
Astfel, vom folosi `ArrayList` dacă avem nevoie de regăsirea unor elemente la poziții diferite în listă, şi `LinkedList` dacă facem multe operații de editare (ștergeri, inserări) în listă. De asemenea, `ArrayList` foloseste mai eficient spaţiul de memorie decât `LinkedList`, deoarece aceasta din urmă are nevoie de o structură de date auxiliară pentru memorarea unui nod.

Metodele principale ale clasei sunt ilustrate în exemplul următor.

Exemplu

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

//adaugare
      a.add("abcd");
      a.add("mn");
      a.add(1,"xyz");//adaugare pe pozitia
      a.add("bcde");
      a.add("abcd");//se pot repeta
```

```
//afisare-ArrayList are suprascrisa metoda toString
      System.out.println(a);

//accesare
      String s=a.get(1);
      System.out.println(s);

//modificarea unui element de la o pozitie specificata
      a.set(1,"abcd");
      System.out.println(a);

//eliminare
       a.remove(2);
       System.out.println(a);

//cautare - folosind pentru comparare metoda equals
       System.out.println("Pozitia primei aparitii abcd "+
a.indexOf("abcd"));
       System.out.println("Pozitia ultimei aparitii abcd "+
a.lastIndexOf("abcd"));
       System.out.println("Exista abcd: "+a.contains("abcd"));

//parcurgere cu for-each
      System.out.print("Afisare cu for-each: ");
      for(String v:a)
            System.out.print(v+" ");
      System.out.println();

//parcurgerea cu Iterator(vom reveni asupra interfetei Iterator)
      System.out.print("Afisare cu iterator: ");
      Iterator<String> is=a.iterator();
      while(is.hasNext())
            System.out.print(is.next()+" ");
     System.out.println();
   }
}

//versiunea 8
      a.forEach((String x)->{System.out.println(x);});
      a.forEach((x)->{System.out.println(x);});
      a.forEach((x)->System.out.println(x));
      a.forEach(System.out::println);
```

Amintim: În ceea ce priveşte căutarea unui obiect în colecţie, pentru compararea a două referinţe se foloseşte metoda `equals`. În clasa `Object` există metoda `equals` şi, cum toate clasele extind direct sau indirect `Object`, ele moştenesc această metodă. Metoda `equals` a clasei `Object` verifică dacă obiectul specificat ca parametru este acelaşi cu cel curent. Într-o clasă putem suprascrie această metodă pentru a defini modul în care se face testul de egalitate pentru obiecte aparţinând acestei clase.

• Algoritmi

Colecțiile pun la dispoziție metode care efectuează diverse operații utile cum ar fi: căutarea, sortarea definite pentru colecţii (pentru obiecte ce implementează interfețe ce descriu colecţii). Aceşti algoritmi se numesc şi *polimorfici* deoarece aceeaşi metodă poate fi folosită pe implementări diferite ale unei colecţii.

Cei mai importanți algoritmi sunt metode **statice** definite în clasa `Collections` și permit efectuarea unor operații utile cum ar fi căutarea, sortarea etc.

Observație: **Algoritmi similari pentru tablouri se găsesc în clasa java.util.Arrays**

Exemplu Să adăugăm în finalul exemplului anterior secvența de cod

```
Collections.sort(a);
System.out.println(a);
```

efectul va fi ordonarea lexicografică a elementelor colecției

Dacă adăugăm apoi și secvența

```
Collections.reverse(a);
System.out.println(a);
```

se inversează ordinea elementelor, acestea devenind ordonate descrescător.

Amintim în plus metodele: `min`, `max`, `binarySearch`, `copy`

Observație Pentru a stabili o ordine între obiecte (un criteriu de comparare pentru metoda sort), elementele colecţiei care urmează să fie sortată trebuie să aibă ca tip o clasă care implementează interfaţa Comparable. Vom reveni asupra acestei interfețe.

TIPURI GENERICE (TIPURI PARAMETRIZATE)

O clasă generică (parametrizată) pot avea unul sau mai mulţi parametri de tip. O astfel de clasă se obţine adăugând un argument pentru fiecare tip parametrizat. Este recomandat ca numele parametrului de tip sa fie o literă mare.

Exemplu

```
class Pereche<T,S>{
      T x; S y;
      Pereche(T x, S y){
            this.x=x;
            this.y=y;
      }
      public String toString(){
            return "("+x+","+y+")"; //apeleaza x.toString
      }
}
```

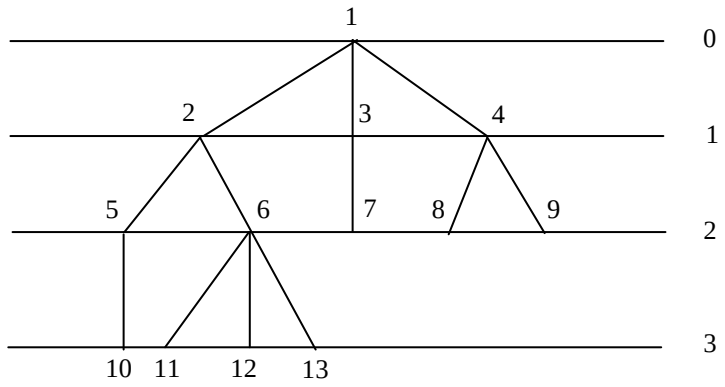
```
class ExpPereche{
    public static void main(String arg[]){
      Pereche<Integer,String> p=new Pereche<Integer,String>(new
Integer(2),"abc");
      System.out.println(p);

      Pereche<String,String> p1=new Pereche<String,String>("abc","d");
      System.out.println(p1);

      Pereche<Double,Integer> p2=new Pereche<Double,Integer>(5.0,4); //nu 5,
trebuie 5.0 - autoboxing
      System.out.println(p2);

      //se poate folosi si fara parametri de tip, ca la colectii
      Pereche pOrice=new Pereche(4,"abc");
      System.out.println(pOrice);
    }
}
```

Vom exemplifica lucrul cu colecții implementând parcurgerea pe niveluri a arborilor oarecare.

ARBORI OARECARE

Primele probleme care se pun sunt aceleaşi ca pentru arborii binari: modalităţile de reprezentare şi de parcurgere.

Considerăm de exemplu următorul arbore



Se consideră că arborele este aşezat pe niveluri şi că pentru fiecare vârf există o ordine între descendenţii săi.

Modul standard de reprezentare al unui arbore oarecare constă în a memora rădăcina, iar pentru fiecare vârf `i` informaţiile:

- `info(i)` = informaţia ataşată vârfului;
- `fiu(i)` = primul vârf dintre descendenţii lui `i`;
- `frate(i)` = acel descendent al tatălui lui `i`, care urmează imediat după `i`.

Lipsa unei legături către un vârf este indicată prin λ.

Pentru arborele din exemplul considerat avem :

```
fiu   =(2,5,7,8,10,11,λ,λ,λ,λ, λ, λ,λ);
frate =(λ,3,4,λ, 6, λ,λ,9,λ,λ,12,13,λ).
```

Observaţie. Prin această reprezentare **fiecărui arbore oarecare i se poate asocia un arbore binar**, identificând `fiu` cu `st` şi `frate` cu `dr`. Această corespondenţă este biunivocă.

O alta reprezentare pentru arborii oarecare sunt ***listele de descendenți***, reprezentare ce include:

- rădăcina `rad`
- pentru fiecare vârf `i`: lista L_i a fiilor vârfului `i`.

O a treia modalitate de reprezentare constă în a memora pentru fiecare vârf tatăl său. Această modalitate este incomodă pentru parcurgerea arborilor, dar se dovedeşte utilă în alte situaţii (ca de exemplu determinarea de lanțuri, reprezentarea mulțimilor disjuncte).

În unele cazuri este util să memorăm pentru fiecare vârf atât fiul şi fratele său, cât şi tatăl său.

- **Parcurgerea în preordine**

Se parcurg recursiv în ordine rădăcina şi apoi subarborii care au drept rădăcină descendenţii săi.

Pentru exemplul considerat avem

1
1,**2**,**3**,**4**
1,2,**5**,**6**,3,7,4,8,9
1,2,5,10,6,11,12,13,3,7,4,8,9.

Concret, executăm apelul `Apreord(rad)` pentru procedura:

```
procedure Apreord(x)
   if x!=λ
      vizit(x); Apreord(fiu(x)); Apreord(frate(x))
end
```

- **Parcurgerea în postordine**

Se parcurg recursiv în ordine subarborii rădăcinii şi apoi rădăcina.

Pentru exemplul considerat:

1
2,**3**,**4**,1
5,**6**,2,7,3,8,9,4,1
10,5,11,12,13,6,2,7,3,8,9,4,1.

Concret, executăm apelul `Apostord(rad)` pentru procedura:

```
procedure Apostord(x)
   if x!=λ
        y←fiu(x);
        while y<>λ
           Apostord(y); y←frate(y)
        vizit(x)
end
```

Observaţie. Parcurgerea în postordine a unui arbore oarecare se reduce la parcurgerea în **inordine** a arborelui binar asociat, deci parcurgerea în postordine a unui arbore oarecare reprezentat folosind fiu-frate se poate face şi astfel:

```
procedure Apostord (x)
   if x!=λ
      Apostord (fiu(x)); vizit(x); Apostord (frate(x))
end
```

Exemplu Parcurgerile în preordine şi postordine a unui arbore reprezentat cu legături fiu-frate.

```
import java.util.*;

class Varf{
      int info;
      Varf fiu,frate;
      Varf () {
      }
      Varf (int i) {
            info = i;
      }
}

class ArbOarecareFF {
  Varf rad;
  static Scanner sc = new Scanner(System.in);

  void creare() {
    System.out.print("rad : ");
    rad = new Varf(sc.nextInt());
    subarb(rad);
  }

  void subarb(Varf x) { //x - deja alocat
    int v;
    // v<0 <==> nu exista
    System.out.print("fiul lui " + x.info + " : ");
    v = sc.nextInt();
    if( v>=0 ) {
       x.fiu = new Varf(v);
       subarb(x.fiu);
    }
    System.out.print("fratele lui " + x.info + ": ");
    v = sc.nextInt();
    if( v>=0 ) {
       x.frate = new Varf(v);
       subarb(x.frate);
    }
  }

  void pre(){
     pre(rad);
  }
```

```
  void pre(Varf x) {
    if( x != null ) {
        System.out.print(x.info + "  ");
        pre(x.fiu);
        pre(x.frate);
    }
  }
  void post(){
     post(rad);
  }
  void post(Varf x) {
    Varf y = x.fiu;
    while (y != null) {
       post(y);
       y = y.frate;
    }
    System.out.print(x.info + " ");
  }
}

class ExpArbOarecareFF {
  public static void main(String[] args) {
    ArbOarecareFF ob = new ArbOarecareFF();
    ob.creare();
    System.out.print("Preordine :\t");
    ob.pre();
    System.out.print("\nPostordine :\t");
    ob.post();
  }
}
```

- **Parcurgerea pe niveluri**

Se parcurg vârfurile în ordinea distanţei lor faţă de rădăcină, ţinând cont de ordinea în care apar descendenţii fiecărui vârf. Pentru exemplul considerat:
1,2,3,4,5,6,7,8,9,10,11,12,13.

Pentru implementare vom folosi o coadă C, în care iniţial apare numai rădăcina. Atâta timp cât coada este nevidă, vom extrage primul element, îl vom vizita şi vom introduce în coadă descendenţii săi:

```
C ← ∅; C ⇐ rad
while C ≠ ∅
   x ⇐ C; vizit(x);
   y ← fiu(x);
   while y ≠ λ
      y ⇒ C; y ← frate(y)
end
```

Parcurgerea pe niveluri este în general utilă atunci când se caută vârful care este cel mai apropiat de rădăcină şi care are o anumită proprietate/informaţie.

Exemplu Exemplificăm în cele ce urmează principalele elemente de limbaj discutate implementând parcurgerea pe niveluri a unui arbore oarecare reprezentat cu liste de descendenți (rădăcina este considerată ca fiind 0)

Varianta 1 – Memorăm listele de descendenți în tablou **bidimensional**

```
import java.util.*;
class ArbOarecare {
  int[][] mat;
  int n;
  void citire() {
    int[] temp;
    int ntemp;
    Scanner sc = new Scanner(System.in);
    System.out.print("n= ");
    n = sc.nextInt();
    mat = new int[n][];
    temp = new int[n];
    for (int i=0; i<n; i++) {
       System.out.print("Fiii lui " + i + " : ");
       ntemp = 0;
       while( sc.hasNextInt() )
           temp[ntemp++] = sc.nextInt();
       sc.next();
       mat[i] = new int[ntemp];
       for (int j=0; j<ntemp; j++)
           mat[i][j] = temp[j];
    }
    System.out.println("**************");
  }
  void parcurgere() {// radacina este varful 0
    int i,j,k;
    ArrayList<Integer> coada = new ArrayList<Integer>();
    coada.add(0); //autoboxing
    while ( ! coada.isEmpty() ) {
       i=coada.get(0);//unboxing
       coada.remove(0);
       System.out.print(i+"  ");
       for (k=0; k<mat[i].length; k++) {
          j = mat[i][k];
          coada.add(j);
       }
    }
  }
}
class ArbNiveluri {
  public static void main(String[] s) {
     ArbOarecare a=new ArbOarecare();
     a.citire();
     a.parcurgere();
  }
}
```

Varianta 2 – Memorăm listele de descendenţi în vector (tablou unidimensional) de liste

```
import java.util.*;

class ArbOarecare {

  ArrayList<Integer>[] mat;
  int n;

  void citire() {
    Scanner sc = new Scanner(System.in);
    System.out.print("n= ");
    n = sc.nextInt();
    mat = new ArrayList[n];  //nu ArrayList<Integer>
    for (int i=0; i<n; i++) {
       System.out.print("Fiii lui " + i + " : ");
       mat[i]=new ArrayList<Integer>();//trebuie alocat
       while( sc.hasNextInt() )
           mat[i].add(sc.nextInt());
       sc.next();
    }
    System.out.println("**************");
  }


  void parcurgere() {// radacina este varful 0
    int i,j,k;
    ArrayList<Integer> coada = new ArrayList<Integer>();
    coada.add(0);
    while ( ! coada.isEmpty() ) {
       i=coada.get(0);
       coada.remove(0);
       System.out.print(i+"  ");
       for (k=0; k<mat[i].size(); k++) {
           j = mat[i].get(k);
           coada.add(j);
       }
    }
  }
}

class ArbNiveluri {
  public static void main(String[] s) {
      ArbOarecare a=new ArbOarecare();
     a.citire();
     a.parcurgere();
  }
}
```

Observaţie. Folosirea clasei `ArrayList` este pur demonstrativă şi nu neapărat necesară. Cum fiecare vârf este înscris exact o dată în coadă, puteam folosi un tablou cu `n` elemente şi memora indicii primului şi ultimului element.