

PROGRAMA

- ▶ **Introducere în limbajul Java**
 - Elemente de bază
 - Tablouri
 - Colecții. Tipuri generice
 - Extinderi
 - Interfețe
 - awt
- ▶ **Introducere în algoritmi**
- ▶ **Tehnici de programare:**
 - Backtracking
 - Greedy
 - Divide et Impera
 - Programare dinamica
 - Branch and Bound
- ▶ **Algoritmi probabiliști. Algoritmi genetici**
- ▶ **Algoritmi nedeterminiști. NP-complexitudine**
- ▶ **Principiul lui Dirichlet**

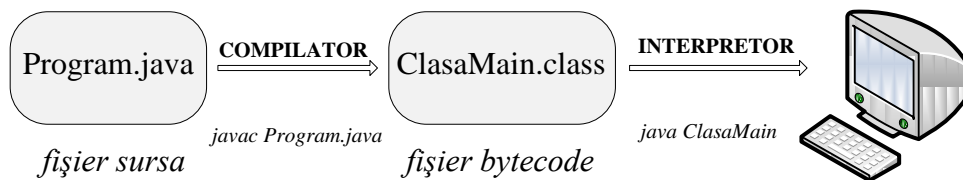
BIBLIOGRAFIE

1. Horia Georgescu. **Tehnici de programare**. Editura Universității din București 2005
2. T.H. Cormen, C.E. Leiserson, R.R. Rivest – **Introducere în algoritmi**, MIT Press, trad. Computer Libris Agora
3. Leon Livovschi, Horia Georgescu. **Sinteza și analiza algoritmilor**. 1986
4. <http://infoarena.ro/>
5. <http://community.topcoder.com/tc>
6. <http://www.oracle.com/technetwork/java/javase/documentation/index.html>
7. Horia Georgescu. **Introducere în universul Java**. Editura Tehnică București 2002
8. Ivor Horton – **Beginning Java 2, Java 7 Edition**, Wiley Pub., 2011
9. Ștefan Tanasă, Cristian Oлару, Ștefan Andrei, **Java de la 0 la expert**, ediția a II-a, Polirom 2007
10. M. Naftalin, P. Wadler - **Java Generics and Collections**, O'Reilly, 2007

Limbajul JAVA

Caracteristici:

- ♦ **Orientat pe obiecte**
- ♦ **Simplitate**
- ♦ **Compilare prealabilă urmată de executare pe mașina gazdă**



- ♦ **Colectorul de reziduuri.** Java permite crearea explicită de obiecte (de tipul unei clase). Distrugerea acestor obiecte este preluată de *colectorul de reziduuri* (*garbage collector*), care marchează obiectele ce nu mai sunt folosite și eliberează spațiul ocupat de ele; eliberarea nu se face neapărat imediat, ci periodic sau atunci când spațiul disponibil curent nu mai poate satisface o nouă cerere de alocare de memorie.
- ♦ **Securitate ridicată.**
 - lipsa pointerilor face ca accesarea unor zone de memorie pentru care accesul nu este autorizat să nu fie posibilă.
 - limbajul obligă programatorul să prevadă acțiunile ce trebuie întreprinse la diferitele erori (numite *excepții*) posibile.
 - se verifică permanent, la executare, valoarea indicelui unui tablou înainte de accesarea componentei respective.
- ♦ **Este proiectat pentru lucru în rețea.**
- ♦ **Posibilitatea lansării mai multor fire de executare.**
- ♦ **Extensibilitate.** Limbajul Java permite includerea de *metode native*, adică de funcții scrise în alt limbaj (de obicei C++). Metodele native sunt legate dinamic la programul Java la momentul executării, rolul lor fiind în principal de a mări viteza de executare pentru anumite secvențe din program.
- ♦ **Applet-uri.** Programele Java se împart în două categorii:
 - programe obișnuite, de sine stătătoare (*stand alone*), numite *aplicații*.
 - *applet*-urile. Este folosit același limbaj, dar diferă modalitatea de lansare în executare. Când utilizatorul vizualizează o pagină Web ce include un applet, mașina la care este conectat transmite applet-ul mașinii gazdă, care este cea pe care se execută applet-ul (se presupune că mașina pe care lucrează utilizatorul are un interpretor Java).

Exemplu Afișarea unui mesaj

```
class Unu{
    public static void main(String arg[]) {
        System.out.println("Prima clasa");
    }
}
```

Compilarea: javac Unu.java

În urma compilării ia naștere fișierul `Unu.class`; mai general, pentru orice unitate de compilare (formată din mai multe clase), va lua naștere pentru fiecare clasă un fișier având numele clasei și extensia `class`). Pentru acest fișier putem comanda executarea sa apelând interpretorul `java.exe` astfel:

Rularea: java Unu

CLASE

Exemplu – la curs

- **Clase**

Clasa este unitatea de programare fundamentală în Java. Orice clasă este formată din **câmpuri, metode și constructori**. Câmpurile și metodele unei clase formează împreună *membrii* acelei clase.

O clasă se definește astfel:

```
[modifier] class NumeClasaDefinita [extends NumeClasa]
[implements NumeInterfete]{
    corp;
}
```

Câmpurile unei clase sunt variabile atașate clasei respective și se declară astfel:

```
[modifieri] tip lista_identificatori;
```

Metodele sunt funcțiile declarate în interiorul clasei:

```
[modifieri] tip_returnat numeMetoda(lista_parametrii);
```

Lista tipurilor parametrilor, în ordinea lor de apariție, formează *signatura* metodei. În aceeași clasă pot apărea mai multe metode cu același nume, dar cu semnături diferite

Clasele sunt considerate tipuri. Entitățile al căror tip este o clasă se numesc *obiecte* ale clasei respective. Se mai spune că obiectele sunt *istanțieri (istanțe)* ale claselor.

Constructorii unei clase seamănă cu metodele, dar numele lor este obligatoriu numele clasei și nu întorc valori.

- Unul dintre constructori este automat invocat la crearea unui obiect de tipul clasei respective.
- Acțiunea constructorilor poate fi oricât de complexă, dar în principal sunt folosiți pentru inițializarea unor câmpuri ale obiectului.
- Ca și pentru metode, pot exista mai mulți constructori, dar cu semnături diferite.

Această posibilitate de a exista mai mulți constructori, respectiv mai multe metode cu același nume, dar cu semnături diferite, poartă numele de **supraîncărcare (overloading)**.

Crearea obiectelor se face cu ajutorul operatorului **new**.

Un obiect poate folosi o **variabilă pentru a păstra o referință către el**. Mai multe variabile pot referi același obiect.

De exemplu, dacă C este o clasă, atunci crearea unui obiect de tipul clasei C se poate face astfel:

1) `new C(...);`

prin care este creat un obiect **anonim** de tipul C;

2) `C ob; //declararea variabilei`
`ob = new C(...); //crearea obiectului`

sau prescurtat:

`C ob = new C(...);`

prin care este creat un obiect de tipul C la care ne putem referi prin variabila `ob`.

La folosirea operatorului **new** se întâmplă mai multe lucruri:

- se creează o nouă instanță a clasei date
- se alocă memorie pentru aceasta
- este invocat constructorul corespunzător (cu aceeași semnătură cu cea din lista de argumente).

Dacă în clasa C nu există vreun constructor (declarat explicit), **se presupune că “există” totuși un constructor fără parametrii, care nu prevede nici o acțiune**. De aceea, în acest caz, la creare trebuie folosită forma **new C()**.

Variabila referință `ob` poate fi folosită pentru accesarea câmpurilor și metodelor clasei. Astfel, dacă `x` este un câmp al clasei, el poate fi referit prin `ob.x`, iar dacă `met` este o metodă a clasei, ea poate fi invocată prin `ob.met(...)`.

O variabilă de tip referință poate avea valoarea **null**, care indică o referință către "**nimic**"; drept urmare variabila nu conține o referință validă, deci **nu poate fi folosită pentru a accesa câmpuri sau invoca metode**.

Gestionarea memoriei se face automat, Java având colector de reziduuri (garbage collector). De aceea nu este nevoie să fie dezalocată memoria ocupată de obiect.

ELEMENTE DE BAZĂ ALE LIMBAJULUI JAVA

▪ Comentarii

Există trei tipuri de comentarii:

- **de sfârșit de linie**: încep cu // și se termină la sfârșitul liniei;
- **generale**: încep cu /* și se termină la prima succesiune */.
- **de documentare**: încep cu /** și se termină la prima succesiune */. Acest tip de comentariu poate fi plasat doar imediat înaintea unei clase, unui membru al unei clase sau a unui constructor. Utilitarul *javadoc* este capabil să colecteze comentariile de documentare din codul sursă al claselor și să le introducă în documente HTML.

▪ Constante (literali) și tipuri primitive

În Java, ca și în C sau C++, o variabilă se poate declara prin tipul ei urmat de nume.

```
[modificatori] tip lista_identificatori;
```

Tipul unei variabile poate fi un tip primitiv sau un tip referință (vectori, clase, interfețe).

Menționăm pentru moment doar următorii modificatori:

- *modificatori de acces* (**public**, **private**, **protected**); dacă nu este specificat nici unul dintre acești modificatori, se consideră că este atașat un modificador implicit;
- **static**;
- **final**.

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

1. **Variabile membre (câmpuri)**, declarate în interiorul unei clase, vizibile pentru toate metodele clasei respective și pentru alte clase, în funcție de nivelul lor de acces.
2. **Variabile locale**, declarate într-o metodă sau într-un bloc de cod, vizibile doar în metoda/blocul respectiv
3. **Parametrii metodelor**, vizibili doar în metoda respectivă
4. **Parametrii de la tratarea excepțiilor**

Enumerăm în continuare *tipurile primitive* (de bază)

- *Tipuri întregi* – `byte`, `short`, `int`, `long`
- *Tipuri în virgulă mobilă* – `float`, `double`
- *Tipul boolean*
- *Tipul char*
- *Tipuri întregi*
 - `byte` (octet) – 1 octet
 - `short` (întreg scurt) – 2 octeți, valoare maximă 32767
 - `int` (întreg) – 4 octeți, valoare maximă $\cong 2 \times 10^9$
 - `long` (întreg lung) – 8 octeți, valoare maximă $\cong 9 \times 10^{18}$

Tip	Nr octeți	Valoare minimă	Valoare maximă
<code>byte</code>	1	-128	127
<code>short</code>	2	-32768	32767
<code>int</code>	4	-2.147.483.648	2.147.483.647
<code>long</code>	8	-9.223.372.036.854.775.808	9.223.372.036.854.775.807

Literali întregi

- *normali (de tip int)* – reprezentați pe 32 de biți
- *lungi (de tip long)* – reprezentați pe 64 de biți, identificați prin faptul că au sufixul 'l' sau 'L'.

```
int x = 456789;

long z1 = 499999999999999L;
long z2 = 49_999_999_999_999L; //-java 7

byte y1 = 456789; //possible loss of precision
byte y2 = 1;
```

Baze :

- **10**
- **16** – prefixul 0x sau 0X
- **8** – prefixul 0
- **2 (din versiunea 7)** – prefixul 0b sau 0B

```
int x1 = 0xA1B;
System.out.println(x1);
int x2 = 0B101; //-java 7
System.out.println(x2);
```

În calcule tipurile `byte` și `short` sunt convertite la `int`.

- **Tipuri în virgulă mobilă**

- **float** – 4 octeți
- **double** – 8 octeți.

Literali reali (în virgulă mobilă)

- **dubli** – reprezentați pe 64 de biți, pentru care scrierea este cea uzuală.
 - numere zecimale ce conțin opțional punctul zecimal și pot fi urmate de un exponent prefixat cu e sau E: 21.0 21. 2.1e1 .21E2
- **normali** – reprezentați pe 32 de biți, au în plus sufixul 'f' sau 'F'.

```
double d = 12345.2;  
float f = 12345.2; //possible loss of precision  
float f = 12345.2f;
```

- **Tipul boolean**

Variabilele de acest tip pot lua doar valorile **true** și **false**.

- **Tipul char**

- Variabilele de acest tip sunt reprezentate pe 16 biți (2 octeți) și pot primi ca valoare orice simbol din codul Unicode.
- O variabilă de tip caracter poate fi folosită oriunde poate apărea o valoare întreagă: este considerat numărul său de ordine în setul de caractere Unicode.
- Un caracter poate fi reprezentat oriunde în textul sursă și printr-o așa numită "secvență Escape", având forma `\uhhhh` sau `\Uhhhh`, unde am notat prin h o cifră hexazecimală
- Secvențele escape pot fi folosite pentru a înlocui caractere speciale sau acțiuni.
Exemple:

<i>Secvența</i>	<i>Utilizare</i>
<code>\b</code>	<i>backspace</i>
<code>\t</code>	<i>tab</i> orizontal
<code>\n</code>	<i>line feed</i> (linie nouă)
<code>\"</code>	ghilimele
<code>\'</code>	apostrof
<code>\\</code>	<i>backslash</i>
<code>\uhhhh</code>	caracter Unicode numărul hhhh (în baza 16)

```
char c='a';  
char c1='\u0061';  
System.out.println(c1);
```

Temă. Scrieți un program în care numele unei variabile conține diacritice. Studiați opțiunea `-encoding` a compilatorului `javac`

▪ Clase înfășurătoare

Pentru fiecare tip primitiv există o clasă corespunzătoare, numită clasă înfășurătoare (**wrapper class**) care pune la dispoziție diverse constante și metode, de exemplu de conversie

- `int` - clasa `Integer`
- `double` - clasa `Double`
- `long` - clasa `Long`
- `short` - clasa `Short`
- `byte` - clasa `Byte`
- `boolean` - clasa `Boolean`
- `char` - clasa `Character`

În clasele înfășurătoare există câmpuri constante (**static final**) pentru $-\infty$ și $+\infty$ și "nu este un număr":

```
Double.NaN  
Double.NEGATIVE_INFINITY  
Double.POSITIVE_INFINITY
```

Pentru conversia din șiruri de caractere în numere se pot folosi metode **statice** de tipul `parseTipNumeric` aflate în clasa „înfășurătoare” corespunzătoare tipului în care vrem să facem conversia:

```
String sir = "123";  
int i = Integer.parseInt(sir);  
double j = Double.parseDouble(sir);  
System.out.println(i);  
System.out.println(j);
```

● *Legătura tip primitiv ⇔ clasă înfășurătoare*

Până la versiunea 5 :

```
int i = 1;  
Integer wi = new Integer(i);  
int j = wi.intValue();  
System.out.println(j);
```

Din versiunea - implicit (**autoboxing / unboxing**)

```
int i = 1;  
Integer wi = i;  
int j = wi;  
System.out.println(j);
```

Observație Nu este indicată folosirea claselor înfășurătoare în operații aritmetice, ci doar în lucrul cu colecții

▪ Operatori

- **Operatori aritmetici** + - * / %

Reguli

- orice valoare ce depășește limita admisă este redusă modulo această limită
- împărțirea întreagă se face prin trunchiere
- operatorul % este definit prin: $(x/y)*y + x\%y == x$

```
double x = 5*2/4+8/6;  
System.out.println(x);
```

- Dacă într-o expresie apar doar variabile de tip short sau byte, acestea sunt convertite la int și apoi se evaluează expresia, rezultatul fiind de tip int

```
byte a = 10, b = 13;  
a = (byte) (a*b);  
System.out.println(a);
```

- Dacă într-o expresie apare o variabilă de tip long, toate variabilele sunt convertite la long

- **Operatorii de incrementare și decrementare** ++ și --, care pot fi aplicați operanzilor numerici (întregi sau în virgulă mobilă) atât în formă prefixată cât și în formă postfixată. Diferența între x++ și ++x constă în faptul că incrementarea este realizată după, respectiv înainte de utilizarea lui x în contextul în care apare

```
double y = 3.5;  
System.out.println(y++);  
System.out.println(++y);  
  
char c = 'a';  
System.out.println(++c);  
System.out.println(c+1);
```

- **Operatorii de atribuire** = += -= /= *= %= <<= >>= >>>= &= |= ^=

- **Operatorii relaționali** > >= == < <= !=

- **Operatori logici**

|| (disjuncția logică, sau) && (conjuncția logică, și) ! (negație)
cu mențiunea că la evaluare se face *scurtcircuitare*

| &

- **Operatori pe biți**

- operatorii binari | (sau) & (și) ^ (xor - sau exclusiv)

1.

```
int f = 0b10001;
System.out.println(f);
System.out.println(f&16);

//testare bit
int mask = 0b100;
if( (f& mask) == 0) //!!!obligatoriu ()
    System.out.println("bitul 3 este 0");
else
    System.out.println("bitul 3 este 1");

//setare bit
f |= mask;
if( (f& mask) ==0) //!!!obligatoriu ()
    System.out.println("bitul 3 este 0");
else
    System.out.println("bitul 3 este 1");
```

2.

```
a ^= b;
b ^= a;
a ^= b;
```

- operatorii de translație (shift): <<, >> (cu propagarea bitului de semn), >>>

```
int b1 = -3;
System.out.println(Integer.toBinaryString(b1));
System.out.println(b1>>1);
System.out.println(b1<<1);

System.out.println(Integer.toBinaryString(b1>>>24));
System.out.println(b1>>>24);
```

Observație: În clasele Integer și Long există metode statice pentru operații pe biți (exemplu: bitCount, highestOneBit)

- **Operatorul condițional ?:**

Acest operator se utilizează în expresii sub forma:

(*cond* ? *e1* : *e2*)

a cărei valoare este *e1* dacă *cond* este true, respectiv *e2* dacă *cond* este false; *cond* trebuie să fie o expresie de tip boolean.

- **Operatori postfix**

- cuprinderea între paranteze a indicilor (cu []);
- operatorul de calificare (.) folosit pentru accesarea membrilor claselor;
- parantezele rotunde folosite pentru specificarea listei de parametri din invocări;
- operatorii postfix de incrementare/decrementare ++ și -- de mai sus.

- **Operatorul de conversie a tipurilor**

`(tip) expresie`

`byte < short < int < long < float < double`

Conversii implicite : de la un tip la altul care îi urmează

- **Operatorul + pentru lucrul cu șiruri**

- este folosit pentru concatenarea șirurilor
- dacă un membru al unei sume este un șir de caractere, atunci are loc o conversie implicită a celorlalți membri ai sumei (devenită acum concatenare) la șiruri de caractere;
- printre membrii sumei pot apărea **și variabile referință!** (fiind apelată metoda `toString`, de care vom discuta ulterior)

```
int u=2,v=4;
System.out.println(u+v+" suma");
System.out.println("suma "+u+v);
```

- **Operatorii pentru referințe**

- accesul la câmpuri (prin calificare);
- invocarea metodelor (prin calificare);
- operatorul de conversie;
- operatorii `==` și `!=`;
- operatorul condițional;
- operatorul **`instanceof`** , folosit în contextul:

`ob instanceof Clasa`

care produce o valoare booleană ce este `true` dacă și numai dacă obiectul `ob` **este diferit de null** și este o instanță a clasei `Clasa` sau poate fi convertit la tipul `Clasa`

```
String sir="abc";
System.out.println(sir instanceof String);
System.out.println(sir instanceof Object);
```

- **Precedența operatorilor**

Ordinea în care are loc efectuarea prelucrărilor determinate de operatori este dată în următorul tabel de priorități ale operatorilor (de la prioritate maximă la prioritate minimă):

- operatorii postfix
- operatorii unari de incrementare/decrementare, operatorii + și - unari, operatorul de negație !
- operatorul **new** de creare de obiecte și cel de conversie: *(tip) expresie*
- operatorii multiplicativi: * / %
- operatorii aditivi: + -
- operatorii relaționali și **instanceof**
- operatorii de egalitate: == !=
- operatorul &
- operatorul |
- conjuncția logică &&
- disjuncția logică ||
- operatorul condițional (? :)
- operatorii de atribuire.

Observații:

- la prioritate egală, operatorii "vecini" acționează conform regulilor de asociativitate prezentate în continuare;
- utilizarea parantezelor este mai puternică decât prioritatea operatorilor. Astfel, spre deosebire de $x+y*z$, în $(x+y)*z$ prima operație care va fi executată este adunarea.

- **Asociativitate**

Regula generală o constituie asociativitatea la stânga. Fac excepție următorii operatori, pentru care este folosită asociativitatea la dreapta:

- operatorii unari;
- operatorii de atribuire.
- operatorul (? :).

Exemple.

1) În expresia $x-y+z$ întâi se va efectua scăderea și apoi adunarea;

2) Instrucțiunea:

$x = y = z = 0;$

este echivalentă cu:

$x = (y = (z = 0));$

și are ca efect atribuirea valorii 0 variabilelor x, y, z .

▪ Instrucțiuni

- **Instrucțiunea compusă** grupează mai multe instrucțiuni prin includerea lor între acolade; ia naștere astfel un *bloc*.

- **Instrucțiunea de declarare** asociază unei variabile un anumit tip și eventual îi atribuie o valoare inițială; variabila devine *locală* celui mai interior bloc care o conține în sensul că există atâta timp cât se execută instrucțiuni ale blocului. O instrucțiune de declarare poate să apară oriunde în interiorul unui bloc.

Înainte de utilizarea lor, variabilele locale trebuie să fi primit valori fie prin inițializare la declarare, fie printr-o instrucțiune de atribuire (în caz contrar va fi semnalată o eroare la compilare).

- **Instrucțiunea de atribuire** conține semnul =, eventual prefixat cu un operator.

- **Instrucțiunea vidă** este formată numai din ; și nu prevede vreo prelucrare.

- **Instrucțiunea prin care este creat un obiect** folosește în acest scop operatorul **new**.

- **Instrucțiunile ce controlează ordinea de executare**, ca de exemplu: **if-else**, **for**, **while**, **do - while**

Din versiunea 5 există o formă a instrucțiunii **for**, pentru obiecte iterabile (**for-each**):

for (*tip identificator : obiect_iterabil*) *instr*;

De exemplu, pentru a afișa elementele unui tablou unidimensional de numere întregi putem folosi una din variantele

```
for(int i=0;i<a.length;i++)
    System.out.print(a[i]+" ");
sau
for(int x:a)
    System.out.print(x+" ");
```

- **Instrucțiunea break.**

- **Instrucțiunea continue**

- **Instrucțiunea switch** evaluează o expresie întreagă, a cărei valoare este folosită pentru a detecta o secvență de instrucțiuni ce urmează a fi executată.

O primă formă a ei este:

```
switch (expresie) {
    case val1: secvență_instrucțiuni1
        . . .
    case valk: secvență_instrucțiunik
    default : secvență_instrucțiuni
}
```

unde:

- tipul expresiei poate fi **char**, **byte**, **short** sau **int**;
- **din versiunea 7 tipul expresiei poate fi și String (și enumerare)**
- val_1, \dots, val_k sunt constante (literali sau câmpuri statice finale inițializate cu expresii constante) de același tip cu al expresiei;
- alternativa **default** este opțională.

1.

```
int i = 2;
switch (i) {
    case 1 : System.out.print("unu ");
    case 2 : System.out.print("doi ");
    case 3 : System.out.println("trei");
}
```

2.

```
String s = "stergere";
switch (s){
    case "stergere":System.out.println("sters");break;
    case "adaugare":System.out.println("adaugat");break;
    default:System.out.println("nimic");
}
```

- **Instrucțiunea return** are una dintre formele:

```
return;
return expresie;
```

TABLOURI UNIDIMENSIONALE

▪ Definiție, declarare

Un tablou `a` poate fi declarat folosind una dintre următoarele modalități:

```
tip[] a;  
tip a[];
```

unde `tip` este tipul componentelor tabloului (poate fi un tip primitiv sau un tip referință).

Declararea unui tablou **nu are drept consecință crearea sa**.

Crearea tabloului a declarat mai sus trebuie făcută explicit, prin:

```
a = new tip[n];
```

unde `n` este o constantă sau o variabilă întreagă ce a primit o valoare strict pozitivă.

Un tablou este un tip referință. Prin creare se obține un obiect de tip tablou (obiect numit prin abuz de limbaj tot tablou).

Componentele tabloului pot fi referite prin `a[i]`, cu `i` luând valori în intervalul `0..n-1`; dacă `i` nu este în acest interval, va fi semnalată o eroare la executare. Lungimea tabloului poate fi referită prin `a.length`.

Declararea și crearea pot fi făcute și simultan:

```
int[] a = new int[10];
```

sau printr-o inițializare efectivă, ca de exemplu:

```
int[] a = {0,3,2,5,1}
```

prin care, evident, `a.length` devine 5.

Câmpul **`length`** al unui (obiect de tip) tablou este un câmp constant (cu modificatorii `public` și `final`) de tip `int`; deci, odată creat, **un obiect tablou nu își poate schimba dimensiunea** (numărul de componente). Pe de altă parte, variabilei referință la tablou `i` se poate asocia o referință la un tablou de același tip.

Exemple

1.

```
int[] a = {1,2,3,4};  
a = new int[20];  
//?? a[0]
```

Noul tablou nu are nici o legătură cu cel vechi.

2.

```
int[] a = {1,2,3,4}, b = {11,12,13}, c;  
c = a; a = b; b = c;
```

Observație O atribuire de genul $b = a$ are altă semnificație decât copierea elementelor lui a în b și nu poate fi folosită în acest scop. Este o **atribuire de referințe**, în urma acestei atribuirii variabilele b și a vor referi același obiect (tablou). Dacă modificăm un element al lui a se modifică și b și invers.

3. Încercați pe rând fiecare dintre cele 3 variante propuse pentru “copierea” elementelor unui vector a în alt vector b . Justificați rezultatele afișate.

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

// Varianta 1 - Nu are efectul dorit
b = a;
System.out.println(a[0]+" "+b[0]);
b[0] = 5;
System.out.println(a[0]+" "+b[0]);
a[0] = 6;
System.out.println(a[0]+" "+b[0]);

// Varianta 2
for(int i=0; i<a.length; i++)
    b[i] = a[i];

// Varianta 3
System.arraycopy(a, 0, b, 0, a.length);
```

În clasa `Arrays` din pachetul `java.util` există metode utile pentru lucrul cu tablouri :

```
fill(int[] a, int fromIndex, int toIndex, int val)
sort(int[] a)
```


TABLOURI MULTIDIMENSIONALE

Tablourile multidimensionale trebuie gândite ca tablouri unidimensionale ale căror elemente sunt tablouri unidimensionale etc. De aceea referirea la un element al unui tablou multidimensional a se face prin:

`a[indice1]...[indicen].`

1.

```
int[][] a = new int[5][3];
```

2.

```
int[][] a = new int[3][];  
a[0] = new int[3];  
a[1] = new int[4];  
a[2] = new int[2];
```

Ia naștere astfel un tablou de forma:

Evident, `a[1].length=4`.

La aceeași structură se poate ajunge și printr-o inițializare efectivă:

```
int[][] a = { {0,1,2}, {1,2,3,4}, {2,3} };
```

care în plus atribuie valori elementelor tabloului.

Dacă în referirea la un element al unui tablou unul dintre indici nu este în intervalul corespunzător, va apărea excepția **IndexOutOfBoundsException**.

Exemplu Afișarea elementelor unui tablou bidimensional cu for-each

```
int x[][]={{1,2,3,7},{4,5},{8}};  
for(int[] linie:x){  
    for(int elem:linie)  
        System.out.printf("%2d",elem);  
    System.out.println();  
}
```

Observație. În Java parametrii se transmit prin valoare

1.

```
class C{
    int a;
    C(){ }
    C(int a1){ a=a1; }

    void afis(){System.out.println(a); }
}
class TestParam{
    static void modif(C ob){
        ob.a++;
    }

    static void modifOb(C ob){
        ob=new C(5);
    }

    static void creste(int x){
        x++;
    }

    public static void main(String arg[]){
        int x=1;
        creste(x);
        System.out.println(x);
        C ob=new C(1); ob.afis();
        modifOb(ob); ob.afis();
        modif(ob); ob.afis();
    }
}
```

2.

```
import java.util.*;
class Tablou {
    static void met(int[] a) {
        a[0] = 7;
        a = new int[5];
        Arrays.fill(a,0,4,1);
    }
    public static void main(String[] s) {
        int[] a = {1,2,3,4};
        for (int i=0 ; i<a.length; i++)
            System.out.print(a[i]+" ");
        System.out.println();
        met(a);
        for (int el:a)
            System.out.print(el+" ");
    }
}
```

CITIREA DE LA TASTATURĂ

O clasă care se poate folosi pentru citirea datelor de tipuri primitive sau `String` este clasa **Scanner** din pachetul `java.util`.

Această clasă se poate folosi pentru citirea din diferite surse: de la tastatură, din fișier, din obiecte de tip `String`, în funcție de tipul obiectului trimis ca parametru constructorului clasei: `InputStream`, `File`, `String`.

În mod predefinit un obiect de tip `Scanner` citește entități delimitate prin caractere albe și apoi încearcă să le interpreteze în modul cerut.

Pentru tipurile primitive de date există metodele `nextByte()`, `nextShort()`, `nextInt()`, `nextLong()`, `nextFloat()`, `nextDouble()`, `nextBoolean()`.

Pentru a testa dacă sunt disponibile valori de anumit tip există metode ca `hasNextInt()`, `hasNextDouble()` etc.

Există și metodele `hasNext()` și `next()` pentru a testa existența unei următoare entități (fără un tip specificat), respectiv pentru citirea următoarei entități (tipul rezultatului întors de metoda `next()` este `String`).

Mai menționăm metodele `nextLine()` și `hasNextLine()` (utile de exemplu dacă dorim să citim un șir de caractere care conține și spații)

Exemplu – la curs

Exemplificăm în cele ce urmează principalele elemente de limbaj discutate implementând parcurgerea arborilor binari folosind reprezentarea arborelui cu tablouri sau cu legături (înlănțuită).

ARBORI BINARI

Exemplu *Parcurgerea arborilor binari reprezentați folosind tablouri.*

```
import java.util.*;

class ArbBinT {

    int rad, nv;
    int[] st,dr; //int st[],dr[];

    void creare() {
        Scanner sc = new Scanner(System.in);
        System.out.print("Nr. varfuri : ");
        nv = sc.nextInt();
        st = new int[nv];
        dr = new int[nv];

        System.out.print("Radacina (numerotare de la 0): ");
        rad = sc.nextInt();

        for (int i=0; i<nv; i++) {
            System.out.print("fii st si dr ai varfului " + i + " (-
1 daca nu exista): ");
            st[i] = sc.nextInt();
            dr[i] = sc.nextInt();
        }
    }

    void pre(){
        pre(rad);
    }

    void pre(int x) {
        if( x>=0 ){
            System.out.print(x + " ");
            pre(st[x]);
            pre(dr[x]);
        }
    }

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

```

void in(int x) {
    if( x>=0 ){
        in(st[x]);
        System.out.print(x + " ");
        in(dr[x]);
    }
}

void post(){
    post(rad);
}

void post(int x) {
    if( x>=0 ){
        post(st[x]);
        post(dr[x]);
        System.out.print(x + " ");
    }
}

}

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

```

Exemplu *Parcurgerea arborilor binari reprezentați folosind legături.*

```

import java.util.*;

class Varf{
    int info;
    Varf st,dr;
    Varf () {
    }
    Varf (int i) {
        info = i;
    }
}

```

```

class ArbBinL {
    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
        // ataseaza subarb. st. si subarb. dr.
        int v;

        // v<0 <=> nu exista descendent
        System.out.print("Desc. stang al lui " + x.info + ": ");
        v = sc.nextInt();
        if( v>=0 ) {
            x.st = new Varf(v);
            subarb(x.st) ;
        }

        System.out.print("Desc. drept al lui " + x.info + ": ");
        v = sc.nextInt();
        if( v>=0 ) {
            x.dr = new Varf(v);
            subarb(x.dr) ;
        }
    }

    void pre() {
        pre(rad);
    }

    void pre(Varf x) {
        if( x != null ) {
            System.out.print(x.info + " ");
            pre(x.st);
            pre(x.dr);
        }
    }

    void in() {
        in(rad);
    }
}

```

```

void in(Varf x) {
    if( x != null ) {
        in(x.st);
        System.out.print(x.info + "  ");
        in(x.dr);
    }
}

void post(){
    post(rad);
}

void post(Varf x) {
    if( x != null ){
        post(x.st);
        post(x.dr);
        System.out.print(x.info + "  ");
    }
}

}

class Exp2ArbBin {
    public static void main(String[] args) {
        ArbBinL ob = new ArbBinL();
        ob.creare();
        System.out.print("Preordine :\t");
        ob.pre();
        System.out.print("\nInordine :\t");
        ob.in();
        System.out.print("\nPostordine :\t");
        ob.post();
    }
}

```

Temă: Modificați programul anterior astfel încât lipsa unui fiu să fie marcată la citire printr-o literă, nu prin valoarea -1

Clasa String

În Java șirurile de caractere sunt **obiecte** ale clasei `String` din pachetul `java.lang`. Un literal de tip `String` este o secvență de caractere între ghilimele

```
String s = "Un sir de caractere";
```

Amintim o serie de **metode** utile în lucrul cu șiruri de caractere din clasa `String`

- **Determinarea lungimii unui șir:** `length`

- **Accesarea unui caracter din șir:** `charAt`

Spre exemplu, următoarea secvență de cod afișează caracterele din șirul `s` câte unul pe linie

```
String s="Un sir de caractere";
for(int i=0;i<s.length();i++)
    System.out.println(s.charAt(i));
```

- **Căutarea unui caracter sau a unui șir de caractere într-un șir:** `indexOf`, `lastIndexOf` (returnează poziția, -1 dacă nu apare)

- `indexOf (int ch)`
- `indexOf (int ch, int index)`
- `indexOf (String str)`
- `indexOf (String str, int index)`

```
String s = "maama";
int poz = s.indexOf('a');
while(poz != -1){
    System.out.printf("%d ",poz);
    poz = s.indexOf('a',poz+1);
}
```

- **Accesarea unui subșir dintr-un șir:** `substring`

```
String s1 = new String("abcdefg");
System.out.println(s1.substring(4)); //efg
System.out.println(s1.substring(1,5));
//de la 1 la 4, nu la 5-> bcde
```

- **Egalitatea a două șiruri:**

Operatorul `==` testează dacă două referințe indică același obiect. În particular, dacă `s1` și `s2` sunt două variabile de tip `String`, atunci `s1==s2` este `true` doar dacă `s1` și `s2` sunt referințe către același șir (nu dacă șirurile sunt egale în sens lexicografic).

 **Exemplu** Următoarea secvență de cod

```
String s1 = "a b", s2;  
char c1 = 'a', c2 = 'b';  
s2 = c1+" "+c2;  
System.out.println(s1+"/"+s2);  
System.out.println(s1==s2);
```

va afișa


a b/a b

false


Pentru a testa egalitatea se folosește metoda `equals` (sau `equalsIgnoreCase` pentru a nu diferenția literele mari de mici):

```
System.out.println(s1.equals(s2));
```

va afișa `true`

 **Observație:** Metoda `equals` este moștenită din clasa `Object` există metoda `equals`. Orice clasă extinde direct sau indirect `Object`, deci moștenește această metodă. Metoda `equals` a clasei `Object` verifică dacă obiectul referit de parametru este același cu cel current (ca și `==`). Într-o clasă putem suprascrie această metodă pentru a defini modul în care se face testul de egalitate pentru obiecte aparținând acestei clase.

- **Compararea a două șiruri lexicografic:** se face cu metoda `compareTo` (sau `compareToIgnoreCase`)
- **Concatenarea de șiruri** se face folosind operatorul `+`. Operatorul `+` este foarte flexibil, permițând concatenarea la un obiect de tip `String` a obiectelor de orice alt tip (**se apelează implicit metoda `toString()`** a obiectului respectiv) sau a unei variabile de tip primitiv. Mai mult, dacă oricare dintre operanzi este de tip `String`, operatorul `+` îi convertește și pe ceilalți la tipul `String`. Rezultatul operației de concatenare este un șir nou (complet distinct de cele concatenate).

 **Exemplu:** Ce afișează următoarea secvență de cod? Modificați această secvență pentru a afișa rezultatul corect.

```
System.out.println("1+1="+1+1);
```

Valoarea unui șir de caractere de tip **String** **nu poate fi modificată după creare** (de exemplu nu se poate modifica un caracter, sau un subșir al șirului). Dacă este necesară și modificarea șirului de caractere se pot utiliza clasele `StringBuilder` sau `StringBuffer` (dacă lucram cu threaduri) din pachetul `java.lang`.

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 definiție 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 *aceeași adresă* în memorie)

Metoda `equals` trebuie să implementeze o **relație de echivalență** (reflexivă, simetrică, tranzitivă). Este recomandabil ca la comparare unui obiect cu `null` sau cu un obiect de alt tip metoda `equals` să întoarcă `false`.

Observație. Metoda `equals` este folosită de metodele de căutare în *colecții* 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 aceeași valoare (**două obiecte egale trebuie să aibă aceeași valoare hash**)
- este recomandabil ca două obiecte diferite să aibă valori hash diferite (în acest fel putând fi îmbunătățită performanța tabelor de dispersie) sau cel puțin ca valorile să fie uniform distribuite într-o plajă de valori

Observație. Este necesar ca suprascrierea metodei `equals` să fie însoțită de suprascrierea metodei `hashCode`, necesitate ce se observă cel mai bine în lucrul cu dicționare (`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 pornește 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 și 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)
```

și pentru variabila `l` se aplică iii:

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

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

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

Numărul 31 a fost ales pentru ca este număr prim impar și pentru că înmulțirea cu 31 se poate optimiza folosind shiftarea pe biți:

```
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 șir de caractere (de obiect din clasa `String`). În clasa `Object` metoda returnează un șir 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)

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 indexOf 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

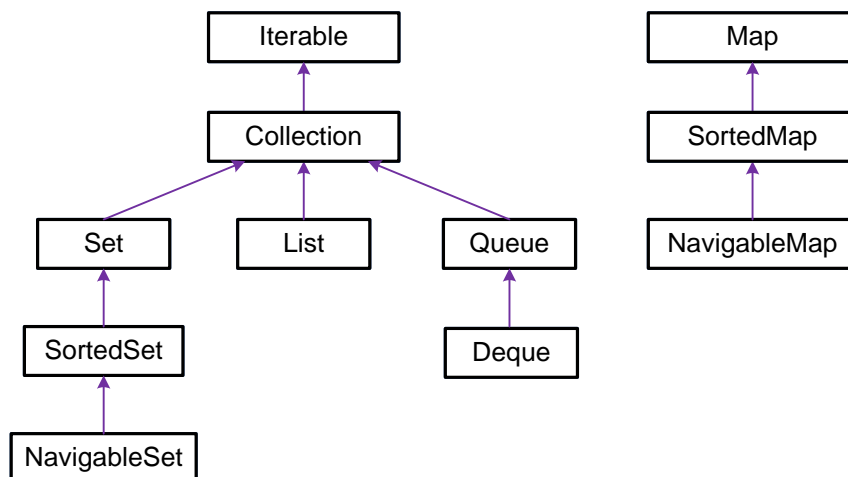
COLECȚII

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

Prin intermediul colecțiilor avem acces la diferite tipuri de date cum ar fi vectori, liste înlanțuite, stive, mulțimi matematice, tabele de dispersie, etc.

Colecțiile sunt folosite atât pentru memorarea și manipularea datelor, cât și pentru transmiterea unor informații de la o metodă la alta.

Menționăm că **interfețele** reprezintă nucleul mecanismului de lucru cu colecții, 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 dispoziție fiecare tip de colecție). Vom înțelege mai bine acest mecanism când vom discuta despre interfețe.



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

Există mai multe tipuri de colecții, corespunzătoare interfețelor amintite:

- **mulțime** (`HashSet`, `TreeSet`) - este o colecție care nu conține duplicate și este menită să abstractizeze noțiunea matematică de mulțime
- **listă**
 - `LinkedList`
 - `ArrayList` (nesincronizat), `Vector` (sincronizat)
 - `Stack`
 - `PriorityQueue`
 - `ArrayDeque` (coadă dublă)

- **dictionare de perechi cheie-valoare/ tabele de dispersie** (`HashMap`, `HashTable`, `IdentityHashMap`, `TreeMap`)- asociază unei *chei* o anumită *valoare*. Funcția nu este neapărat injectivă, deci mai multor chei li se poate asocia aceeași valoare. Atât cheile, cât și 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, performanțe...) se pot găsi în cartea **M. Naftalin, P. Wadler - Java Generics and Collections, O'Reilly, 2007**

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

- au definită metoda `toString`, care returnează o reprezentare ca șir de caractere a colecției respective
- permit crearea de **iteratori** pentru parcurgerea tuturor elementelor colecției unul câte unul (implementează interfața `Iterable`), cu excepția dicționarelor, și de aceea pentru ele este posibilă utilizarea instrucțiunii `for` de tip „for each” (vezi exemplele de mai jos)
- au atât constructor fără argumente cât și un constructor care acceptă ca argument o altă colecție

Dar ce tip de obiecte poate conține o colecție?

Până la versiunea 5 nu se putea specifica tipul obiectelor introduse într-o colecție, o colecție de date punând conține obiecte de orice tip (chiar de tipuri diferite, ele fiind considerate de tip `Object`). Chiar dacă obiectele conținute în colecție erau de același tip, nu exista o modalitate ca acesta să fie asociat colecției. Era în sarcina programatorului să știe ce tipuri au obiectele introduse în colecție, 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 colecții era `Object`). Dacă programatorul greșeste tipul elementului, acest lucru nu se poate verifica la compilare, existând riscul apariției unei excepții la execuție 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)+" ");

        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 sursă cu o versiune mai veche a JDK-ului pentru a vedea ce era permis și ce nu în versiunile mai vechi puteți folosi opțiunea `-source` a comenzii `javac`, de exemplu:

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

Se poate lucra însă și 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);//apeleaza toString,
scrie toata colectia

//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 colecție nu mai este necesară conversia explicită la extragerea unui element.
2. În cazul folosirii tipurilor generice, încercarea de a utiliza în cadrul unei colecții a unui element necorespunzător ca tip va produce o **eroare la compilare**, spre deosebire de varianta anterioară ce permitea doar aruncarea unei excepții la execuție de tipul `ClassCastException` în cazul folosirii incorecte a tipurilor
3. Faptul că anumite clase sunt parametrizate este utilizat **doar de compilatorul Java** nu și de mașina virtuală, codul generat de compilatorul Java putând fi executat și de versiunea 1.4 a platformei.

• **Liste. Clasa ArrayList**

`ArrayList` și `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
<code>ArrayList</code>	1	1	n	1	n
<code>LinkedList</code>	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 înlanțuită necesitând parcurgerea secvențială a listei până 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, și `LinkedList` dacă facem multe operații de editare (ștergeri, inserări) în listă. De asemenea, `ArrayList` folosește mai eficient spațiul 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"));

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

//parcurserea 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();
    }
}

```

Amintim: În ceea ce privește căutarea unui obiect în colecție, pentru compararea a două referințe se folosește metoda `equals`. În clasa `Object` există metoda `equals` și, cum toate clasele extind direct sau indirect `Object`, ele moștenesc această metodă. Metoda `equals` a clasei `Object` verifică dacă obiectul specificat ca parametru este același cu cel curent. Într-o clasă putem suprascrie această metodă pentru a defini modul în care se face testul de egalitate pentru obiecte aparținând acestei clase.

• Algoritmi

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

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 colecției care urmează să fie sortată trebuie să aibă ca tip o clasă care implementează interfața `Comparable`. Vom reveni asupra acestei interfețe.

TIPURI GENERICE (TIPURI PARAMETRIZATE)

O clasă generică (parametrizată) pot avea unul sau mai mulți parametri de tip. O astfel de clasă se obține adăugând un argument pentru fiecare tip parametrizat. Este recomandat ca numele parametrului de tip să 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);

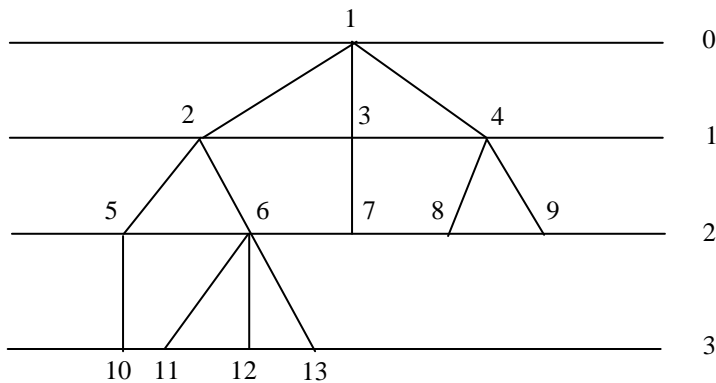
        //se poate folosi si fara parametri de tip, ca la colectii
        Pereche pOrice=new Pereche(4,"abc");
        System.out.println(pOrice);
    }
}
```

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

ARBORI OARECARE

Primele probleme care se pun sunt aceleași ca pentru arborii binari: modalitățile de reprezentare și de parcurgere.

Considerăm de exemplu următorul arbore



Se consideră că arborele este așezat pe niveluri și că pentru fiecare vârf există o ordine între descendenții săi.

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

- $\text{info}(i)$ = informația atașată vârfului;
- $\text{fiu}(i)$ = primul vârf dintre descendenții lui i ;
- $\text{frate}(i)$ = 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 :

$\text{fiu} = (2, 5, 7, 8, 10, 11, \lambda, \lambda, \lambda, \lambda, \lambda, 8, \lambda);$
 $\text{frate} = (\lambda, 3, 4, \lambda, 6, \lambda, \lambda, 9, \lambda, \lambda, 12, 13, \lambda).$

Observație. Prin această reprezentare **fiecărui arbore oarecare i se poate asocia un arbore binar**, identificând fiu cu st și frate cu dr . Această corespondență 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 dovedește utilă

în alte situații (ca de exemplu determinarea de lanțuri, reprezentarea mulțimilor disjuncte).

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

▪ **Parcurgerea în preordine**

Se parcurg recursiv în ordine rădăcina și apoi subarborii care au drept rădăcină descendenții 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
  then
  else vizit(x); Apreord(fiu(x)); Apreord(frate(x))
end
```

▪ **Parcurgerea în postordine**

Se parcurg recursiv în ordine subarborii rădăcinii și 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
  then
  else y←fiu(x);
    while y<>λ
      Apostord(y); y←frate(y)
    vizit(x)
end
```

Observație. 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 și astfel:

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

Exemplu Parcurgerile în preordine și 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.create();
        System.out.print("Preordine :\t");
        ob.pre();
        System.out.print("\nPostordine :\t");
        ob.post();
    }
}

```

▪ Parcurgerea pe niveluri

Se parcurg vârfurile în ordinea distanței lor față de rădăcină, ținând cont de ordinea în care apar descendenții 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 inițial apare numai rădăcina. Atâta timp cât coada este nevidă, vom extrage primul element, îl vom vizita și vom introduce în coadă descendenții 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 și care are o anumită proprietate/informație.

Exemplu Prezentăm programul care realizează parcurgerea pe niveluri a unui arbore oarecare reprezentat la 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 descendenți 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();
    }
}

```

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