EVALUARE

- **50% laborator + 50% examen scris**
- **Laborator**
  - **1/2 activitate + 1/2 test în ultima săptămână**
- **Condiții:**
  - **Nota test laborator ≥ 5**
  - **Nota laborator ≥ 5**

PROGRAMA

- **Introducere în limbajul Java**
  - Elemente de bază
  - Colecții. Tipuri generice
  - Aplicaţii – arbori, grafuri
  - Extinderi, interfețe
- **Introducere în algoritmi**
- **Tehnici de programare:**
  - Greedy
  - Divide et Impera
  - Programare dinamica
  - Backtracking
  - Branch and Bound
- **Algoritmi euristici. Algoritmi probabiliști. Algoritmi genetici**
- **Principiul lui Dirichlet**

BIBLIOGRAFIE

1. Jon Kleinberg, Éva Tardos, **Algorithm Design**, Addison-Wesley 2005 http://www.cs.princeton.edu/~wayne/kleinberg-tardos/
2. Horia Georgescu. **Tehnici de programare**. Editura Universităţii din Bucureşti 2005
3. S. Dasgupta, C.H. Papadimitriou, U.V. Vazirani, **Algorithms**, McGraw-Hill, 2008
4. T.H. Cormen, C.E. Leiserson, R.R. Rivest – **Introducere în algoritmi**, MIT Press, trad. Computer Libris Agora
5. Leon Livovschi, Horia Georgescu. **Sinteza şi analiza algoritmilor**. 1986
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. Ştefan Tanasă, Cristian Olaru, Ştefan 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

Principalele surse pentru materialele prezentate la curs sunt [1], [2], [7], [8].

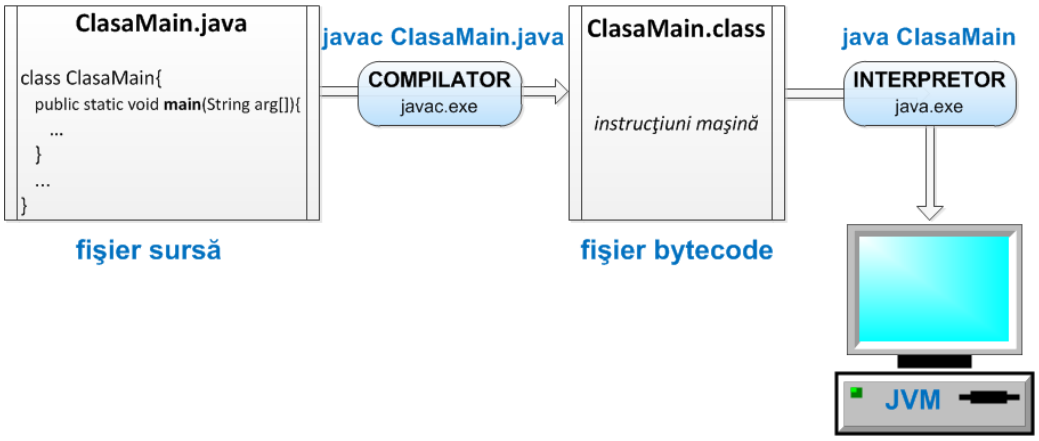
Limbajul JAVA

Cuprins curs

Limbajul JAVA

Caracteristici:

- **Orientat pe obiecte**
- **Simplitate**
- **Compilare prealabilă urmată de executare (interpretare) pe maşina 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 şi eliberează spaţiul ocupat de ele; eliberarea nu se face neapărat imediat, ci periodic sau atunci când spaţiul 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 acţiunile ce trebuie întreprinse la diferitele erori (numite *excepţii*) posibile.
  - se verifică permanent, la executare, valorea indicelui unui tablou înainte de accesarea componentei respective.
- **Este proiectat pentru lucru în reţea.**
- **Posibilitatea lansării mai multor fire de executare.**
- **Extensibilitate.** Limbajul Java permite includerea de *metode native*, adică de funcţii 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 secvenţe din program.
- **Applet-uri.** Programele Java se împart în două categorii:
  - programe obişnuite, de sine stătătoare (*stand alone*), numite *aplicaţii*.
  - *applet*-urile. Este folosit acelaşi limbaj, dar diferă modalitatea de lansare în executare. Când utilizatorul vizualizează o pagină Web ce include un applet, maşina la care este conectat transmite applet-ul maşinii gazdă, care este cea pe care se execută applet-ul (se presupune că maşina pe care lucrează utilizatorul are un interpretor Java).

Un prim exemplu:

Exemplu **Afişarea unui mesaj**

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

Salvăm codul într-un fişier cu numele Unu.java (codul se poate scrie în orice editor şi se salvează cu extensia java)

Compilarea: `javac Unu.java`

În urma compilării ia naştere fişierul `Unu.class`; mai general, pentru orice unitate de compilare (formată din mai multe clase), va lua naştere pentru fiecare clasă un fişier având numele clasei şi extensia `class`). Pentru acest fişier putem comanda executarea sa apelând interpretorul `java.exe` astfel (pentru clasa care conţine metoda main)

Rularea: `java Unu`

ELEMENTE DE BAZĂ ALE LIMBAJULUI JAVA

▪ Comentarii

Există trei tipuri de comentarii:

- ***de sfârşit de linie***: încep cu `//` şi se termină la sfârşitul liniei;

- ***generale***: încep cu `/*` şi se termină la prima succesiune `*/`.

- ***de documentare***: încep cu `/**` şi 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 şi să le introducă în documente HTML.

▪ Constante (literali), variabile şi tipuri primitive

În Java, ca şi 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 referinţă (vectori, clase, interfeţe). Menţionăm pentru moment doar următorii modificatori:

- *modificatori de acces* (**`public, private, protected`**); dacă nu este specificat nici unul dintre aceşti modificatori, se consideră că este ataşat un modificator implicit;
- **`static`**;
- **`final`**.

În funcţie 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 si pentru alte clase, în funcţie 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 excepţiilor**

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 $\cong 2\times10^9$
  - **`long`** (întreg lung) – 8 octeți, valoare maximă $\cong 9\times10^{18}$

Tip	Nr octeţi	***Valoare minimă***	***Valoare maximă***
`byte`	1	-128	127
`short`	2	-32768	32767
`int`	4	-2.147.483.648	2.147.483.647
`long`	8	-9.223.372.036.854.775.808	9.223.372.036.854.775.807

Literali întregi

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

```
int x = 456789;
long z1 = 49999999999999L;
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` şi `short` sunt convertite la `int`.

- ***Tipuri în virgulă mobilă***
  - **`float`** – 4 octeți
  - **`double`** – 8 octeți.

Literali reali (în virgulă mobilă)

- ***dubli*** – reprezentaţi pe 64 de biţi, pentru care scrierea este cea uzuală.
  - numere zecimale ce conţin opţional punctul zecimal şi pot fi urmate de un exponent prefixat cu e sau E: 21.0 21. 2.1e1 .21E2
- ***normali*** – reprezentaţi pe 32 de biţi, 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`** şi **`false`**.

- ***Tipul* `char`**
  - Variabilele de acest tip sunt reprezentate pe 16 biţi (2 octeţi) şi 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 şi printr-o aşa numită "secvenţă Escape", având forma **`\uhhhh`** sau **`\Uhhhh`**, unde am notat prin `h` o cifră hexazecimală
  - Secvenţele escape pot fi folosite pentru a înlocui caractere speciale sau acţiuni.

  Exemple:

Secvenţa	***Utilizare***
\b	*backspace*
\t	*tab* orizontal
\n	*line feed* (linie nouă)
\"	ghilimele
\'	apostrof
\\	*backslash*
\uhhhh	caracter Unicode numărul **hhhh** (în baza 16)

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

Operatori

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

Reguli

- orice valoare ce depăşeşte limita admisă este redusă modulo această limită
- împărţirea 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` şi 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 variabliă de tip `long`, toate variabilele sunt convertite la `long`

- ***Operatorii de incrementare şi decrementare*** `++` şi `--`, care pot fi aplicaţi operanzilor numerici (întregi sau în virgulă mobilă) atât în formă prefixată cât şi în formă postfixată. Diferenţa între x++ şi ++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 relaţionali*** `> >= == < <= !=`

- ***Operatori logici***

  `||` (disjuncţia logică, sau) `&&` (conjuncţia logică, şi) `!` (negaţie)

  cu menţiunea că la evaluare se face ***scurtcircuitare***

- ***Operatori pe biţi***

  - operatorii binari `|` (sau) `&` (şi) `^` (xor - sau excusiv)

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 translaţie (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);
```

Observaţie: În clasele înfăşurătoare `Integer` şi `Long` există metode statice pentru operaţii pe biţi (exemplu: `bitCount`, `highestOneBit`); vom reveni asupra acestor clase

- ***Operatorul condiţional*** **`? :`**

  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 ++ şi −− 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 şiruri***

  - este folosit pentru concatenarea şirurilor
  - dacă un membru al unei sume este un şir de caractere, atunci are loc o conversie implicită a celorlalţi membri ai sumei (devenită acum concatenare) la şiruri de caractere;

- printre membrii sumei pot apărea **şi variabile referinţă**! (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 referinţe***

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

```
ob instanceof Clasa
```

care produce o valoare booleană ce este `true` dacă şi numai dacă obiectul `ob` **este diferit de `null`** şi este o instanţă 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);
```

- **Precedenţa operatorilor**

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

- operatorii postfix
- operatorii unari de incrementare/decrementare, operatorii + şi – unari, operatorul de negaţie `!`
- operatorul **`new`** de creare de obiecte şi cel de conversie: `(tip) expresie`
- operatorii multiplicativi: `* / %`
- operatorii aditivi: `+ –`
- operatorii de translaţie (shift)
- operatorii relaţionali şi **`instanceof`**
- operatorii de egalitate: `== !=`
- operatorul `&`
- operatorul `¦`
- conjuncţia logică `&&`
- disjuncţia logică `¦¦`
- operatorul condiţional `( ? : )`
- operatorii de atribuire.

Observaţii:

- la prioritate egală, operatorii "vecini" acţionează 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 operaţie care va fi executată este adunarea.

- **Asociativitate**

Regula generală o constituie asociativitatea la stânga. Fac excepţie 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 şi apoi adunarea;

2) Instrucţiunea:

```
x = y = z = 0;
```

este echivalentă cu:

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

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

▪ Instrucţiuni

- ***Instrucţiunea compusă*** grupează mai multe instrucţiuni prin includerea lor între acolade; ia naştere astfel un *bloc*.

- ***Instrucţiunea de declarare*** asociază unei variabile un anumit tip şi eventual îi atribuie o valoare iniţială; variabila devine *locală* celui mai interior bloc care o conţine în sensul că există atâta timp cât se execută instrucţiuni ale blocului. O instrucţiune de declarare poate să apară oriunde în interiorul unui bloc.

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

- ***Instrucţiunea de atribuire*** conţine semnul =, eventual prefixat cu un operator.

- ***Instrucţiunea vidă*** este formată numai din `;` şi nu prevede vreo prelucrare.

- ***Instrucţiunea prin care este creat un obiect*** foloseşte în acest scop operatorul **`new`**.

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

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

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

De exemplu, pentru a afişa 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+" ");
```

- ***Instrucţiunea*** **`break`**.

- ***Instrucţiunea*** **`continue`**

- ***Instrucţiunea*** **`switch`** evaluează o expresie întreagă, a cărei valoare este folosită pentru a detecta o secvenţă de instrucţiuni ce urmează a fi executată.

O primă formă a ei este:

```
switch (expresie) {
  case val1: secvenţă_instrucţiuni1
  . . .
  case valk: secvenţă_instrucţiunik
  default   : secvenţă_instrucţiuni
}
```

unde:

- tipul expresiei poate fi **`char`**, **`byte`**, **`short`** sau **`int`**;
- **din versiunea 7 tipul expresiei poate fi şi String**
- val_1, ..., val_k sunt constante (literali sau câmpuri statice finale iniţializate cu expresii constante) de acelaşi tip cu al expresiei;
- alternativa **`default`** este opţională.

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");
}
```

- ***Instrucţiunea*** **`return`** are una dintre formele:

```
return;
return expresie;
```

CLASE

Exemplu - la curs

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

O clasă se defineşte astfel:

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

Câmpurile unei clase sunt variabile ataşate clasei respective şi se declară astfel:

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

Metodele sunt funcţiile declarate în interiorul clasei:

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

Lista tipurilor parametrilor, în ordinea lor de apariţie, formează ***signatura*** metodei. În aceeaşi clasă pot apărea mai multe metode cu acelaşi nume, dar cu signaturi diferite

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

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

- Unul dintre constructori este automat invocat la crearea unui obiect de tipul clasei respective.
- Acţiunea constructorilor poate fi oricât de complexă, dar în principal sunt folosiţi pentru iniţializarea unor câmpuri ale obiectului.
- Ca şi pentru metode, pot exista mai mulţi constructori, dar cu signaturi diferite.

Această posibilitate de a exista mai mulţi constructori, respectiv mai multe metode cu acelaşi nume, dar cu signaturi diferite, poartă numele de ***supraîncărcare* (*overloading*).**

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

Declararea şi crearea unui obiect de tipul `C`

```
C ob1; //variabila referinta – poate memora o referinta la un obiect de tip C
ob1 = new C();
```

sau

```
C ob2 = new C(1,true);
```

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

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

Dacă în clasa C nu există vreun constructor (declarat explicit), **se presupune că "există" totuşi un constructor fără parametri, care nu prevede nici o acţiune**. De aceea, în acest caz, la creare trebuie folosită forma **`new C()`**.

Invocarea metodelor:

```
int i = ob.met(2);
ob.met();
```

Observaţii.

- O variabilă de tip referinţă poate avea valoarea **null**, care indică o referinţă către "**nimic**"; drept urmare variabila nu conţine o referinţă 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.
- La invocarea metodelor se foloseşte *apelul prin valoare*
- Dacă metoda este declarată cu modificatorul `static`, ea poate fi invocată şi prin numele clasei. Astfel, dacă metoda `met` cu signatura vidă era declarată prin:

  ```
  static int met() { . . . }
  ```

  atunci ea putea fi invocată şi prin:

  ```
  C.met();
  ```

- Dacă un câmp `w` este declarat cu modificatorul `static`, el este comun tuturor obiectelor de tipul `C`, deci este memorat o singură dată (joacă rolul de memorie comună pentru toate obiectele ce instanţiază clasa). În plus el poate fi referit şi prin `C.w`

  Exemplu – câmpuri şi metode statice

```
class ExpStatic{
    static int nr=1;
    int x=1;
    static void cresteNr(){  nr++; }
    void cresteX(){  x++;       } //NU static
    void afis(){     System.out.println(nr+" "+x ); }

    public static void main(String s[]){
          ExpStatic.cresteNr();
          System.out.println(ExpStatic.nr);
          ExpStatic ob1,ob2;
          ob1=new ExpStatic();
          ob2=new ExpStatic();
          ob1.cresteNr();
          ob1.cresteX();
          ob1.afis();
          ob2.afis();
    }
}
```

TABLOURI UNIDIMENSIONALE

- **Definiţie, declarare**

Un tablou a poate fi declarat folosind una dintre următoarele modalităţi:

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

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

Declararea unui tablou **nu are drept consecinţă 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.

Exemplu

```
int a[];
a = new int[5];
a[0] = 1;
```

Un tablou este un tip referinţă. Prin creare se obţine 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 şi crearea pot fi făcute şi simultan:

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

sau printr-o iniţializare 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` şi `final`) de tip `int`; deci, odată creat, **un obiect tablou nu îşi poate schimba dimensiunea** (numărul de componente). Pe de altă parte, variabilei referinţă la tablou i se poate asocia o referinţă la un tablou de acelaşi 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;
```

Observaţie O atribuire de genul `b = a` are altă semnificaţie decât copierea elementelor lui `a` în `b` şi nu poate fi folosită în acest scop. Este o **atribuire de referinţe**, în urma acestei atribuiri variabilele `b` şi `a` vor referi acelaşi obiect (tablou). Dacă modificăm un element al lui a se modifică şi `b` şi 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;

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

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

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]);
```

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

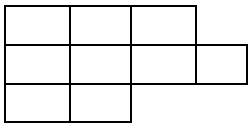
Tablourile multidimensionale trebuie gândite ca tablouri unidimensionale ale căror elemente sunt tablouri unidimensionale etc. De aceea referirea la un element al unui tablou multidimensional `a` se face prin:

```
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 naştere astfel un tablou de forma:



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

La aceeaşi structură se poate ajunge şi printr-o iniţializare 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 excepţia **`IndexOutOfBoundsException`**.

Exemplu **Afişarea 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("%4d",elem);
    System.out.println();
}
```

CLASE ÎNFĂŞURĂTOARE

Pentru fiecare tip primitiv există o clasă corespunzătoare, numită clasă înfăşurătoare (**wrapper class**) care pune la dispoziţie diverse constante şi 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 înfăşurătoare există câmpuri constante (**`static final`**) pentru -∞ şi +∞ şi "nu este un număr":

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

Pentru conversia din şiruri de caractere în numere se pot folosi metode **statice** de tipul `parse`*`TipNumeric`* aflate în clasa „înfăşurătoare” 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 înfăşurătoare***

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);
```

Obserervaţie **Nu este indicată folosirea claselor înfăşurătoare în operaţii aritmetice, ci doar în lucrul cu colecţii**

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 fişier, din obiecte de tip `String`, în funcţie de tipul obiectul trimis ca parametru constructorului clasei: `InputStream`, `File`, `String`.

În mod predefinit un obiect de tip `Scanner` citeşte entităţi delimitate prin caractere albe şi 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 şi metodele `hasNext()` şi `next()` pentru a testa existenţa unei următoare entităţi (fără un tip specificat), respectiv pentru citirea următoarei entităţi (tipul rezultatului întors de metoda `next()` este `String`).

Mai menţionăm metodele `nextLine()` şi `hasNextLine()` (utile de exemplu dacă dorim să citim un şir de caractere care conţine şi spaţii)

Exemplu – la curs

TRANSMITEREA PARAMETRILOR

După cum aminteam, în Java parametrii se transmit **prin valoare**. Pentru a înţelege ce înseamnă acest lucru şi în cazul variabilelor referinţă considerăm exemplele următoare.

1.
```
class C{
      int a;
      C(){  }
      C(int a1){ a=a1; }

      void afis(){System.out.println(a); }
}

class TestParam{
      static void modif(C ob){ //modific camp
            ob.a++;
      }

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

      static void creste(int x){ //modific variabila de tip primitiv
            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+" ");
  }
}
```

Exemplificăm în cele ce urmează principalele elemente de limbaj discutate implementând parcurgerea arborilor binari folosind reprezentarea arborelui cu tablouri sau cu legături (înlănțuită). **Pe masură ce vom prezenta și alte elemente de limbaj, clasele pot fi reproiectate mai bine (conform principiilor POO).**

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 Exp1ArbBin {
  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: **Modificaţi programul anterior astfel încât lipsa unui fiu să fie marcată la citire printr-o literă, nu printr-o valoare întreagă negativă**