**REFERINȚE**

O **referință** reprezintă o modalitate de accesare a unei zone de memorie alocată dinamic (i.e., în zona de heap) prin intermediul adresei sale. Deși o referință utilizează adresa de memorie a unui obiect pentru a-l accesa, acest lucru nu se realizează într-un mod transparent pentru programator, respectiv acesta nu poate determina adresa respectivă și nici nu poate executa operații specifice pointerilor din limbajele C/C++!

Referințele, la fel ca variabilele de tip primitiv, sunt memorate în zona de stivă (stack). Practic, diferența dintre o variabilă de tip primitiv și una de tip referință constă în faptul că o variabilă de tip primitiv conține direct o valoare de un anumit tip, în timp ce o variabilă de tip referință conține doar "adresa" din zona de heap la care se află alocat un anumit obiect:

|  |  |  |
| --- | --- | --- |
|  | **STACK** | **HEAP** |
| **int x = 10;** | **10**  **x** |  |
| **Persoana p = new Persoana();** | **Referința obiectului**  **p** | **Obiect de tip Persoana** |

Efectuarea unei operații de atribuire între două referințe NU va conduce la realizarea unei copii a obiectului referit, ci obiectul respectiv va putea fi accesat prin intermediul ambelor referințe, ceea ce poate conduce la efecte colaterale nedorite:

|  |  |  |
| --- | --- | --- |
|  | **STACK** | **HEAP** |
| **Persoana p = new Persoana();** | **Referința obiectului**  **p** | **Obiect de tip Persoana** |
| **Persoana q = p;** | **Referința obiectului**  **q** |

Literalul null este utilizat pentru a indica faptul că o referință nu este asociată cu niciun obiect.

**TABLOURI**

În limbajul Java, tablourile sunt considerate variabile de tip referință, deci ele trebuie inițializate sau alocate dinamic înainte de a fi utilizate.

Declararea tablourilor unidimensionale (de fapt, a unor referințe spre tablouri unidimensionale!) se poate realiza în două moduri:

1. **tip\_de\_date[] tablou\_1, tablou\_2, ..., tablou\_n;**

De exemplu, prin **int[] a, b;** se vor declara două referințe **a** și **b** spre tablouri unidimensionale cu elemente de tip **int** care trebuie ulterior să fie alocate dinamic!

1. **tip\_de\_date tablou\_1[], tablou\_2[], ..., tablou\_n[];**

De exemplu, prin **int a[], b;** doar variabila **a** este o referință spre un tablou unidimensional cu elemente de tip **int**, iar variabila **b** este una de tipul primitiv **int**!

Tablourile unidimensionale pot fi inițializate în momentul declarării lor folosind un șir de valori, astfel:

1. **tip\_de\_date[] tablou = {valoare\_1, ..., valoare\_n};**

**Exemplu:** **int**[] a = {1, 2, 3, 4, 5}, b = {10, 20, 30};

1. **tip\_de\_date tablou[] = {valoare\_1, ..., valoare\_n};**

**Exemplu:** **int** a[] = {1, 2, 3, 4, 5}, b[] = {10, 20, 30};

Alocarea dinamică a tablourilor unidimensionale se realizează folosind operatorul **new**, astfel:

**tablou = new tip\_de\_date[număr\_elemente];**

**Exemple:**

|  |  |  |
| --- | --- | --- |
| **int** a[];  ..............  a = **new int**[5]; | **int**[] a = **null**;  ..............  a = **new int**[5]; | **int** a[] = **new int**[5];  **int** []b = **new int**[7]; |

Toate elementele unui tablou alocat dinamic vor fi inițializate cu valori nule de tip!

După inițializarea sau alocarea dinamică a unui tablou, data membră **length** va conține dimensiunea fizică a tabloului, adică numărul maxim de elemente pe care le poate conține tabloul.

Parcurgerea unui tablou se poate realiza fie pozițional (prin intermediul indicilor elementelor), fie folosind o instrucțiune repetitivă de tipul **enhanced-for:**

**int**[] a = {10, 20, 30, 40, 50};

System.out.println(a.length); **//se va afișa valoarea 5**

**for**(**int** i = 0; i < a.length; i++) **//se vor afișa valorile 10 20 30 40 50**

System.out.print(a[i] + " ");

System.out.println();

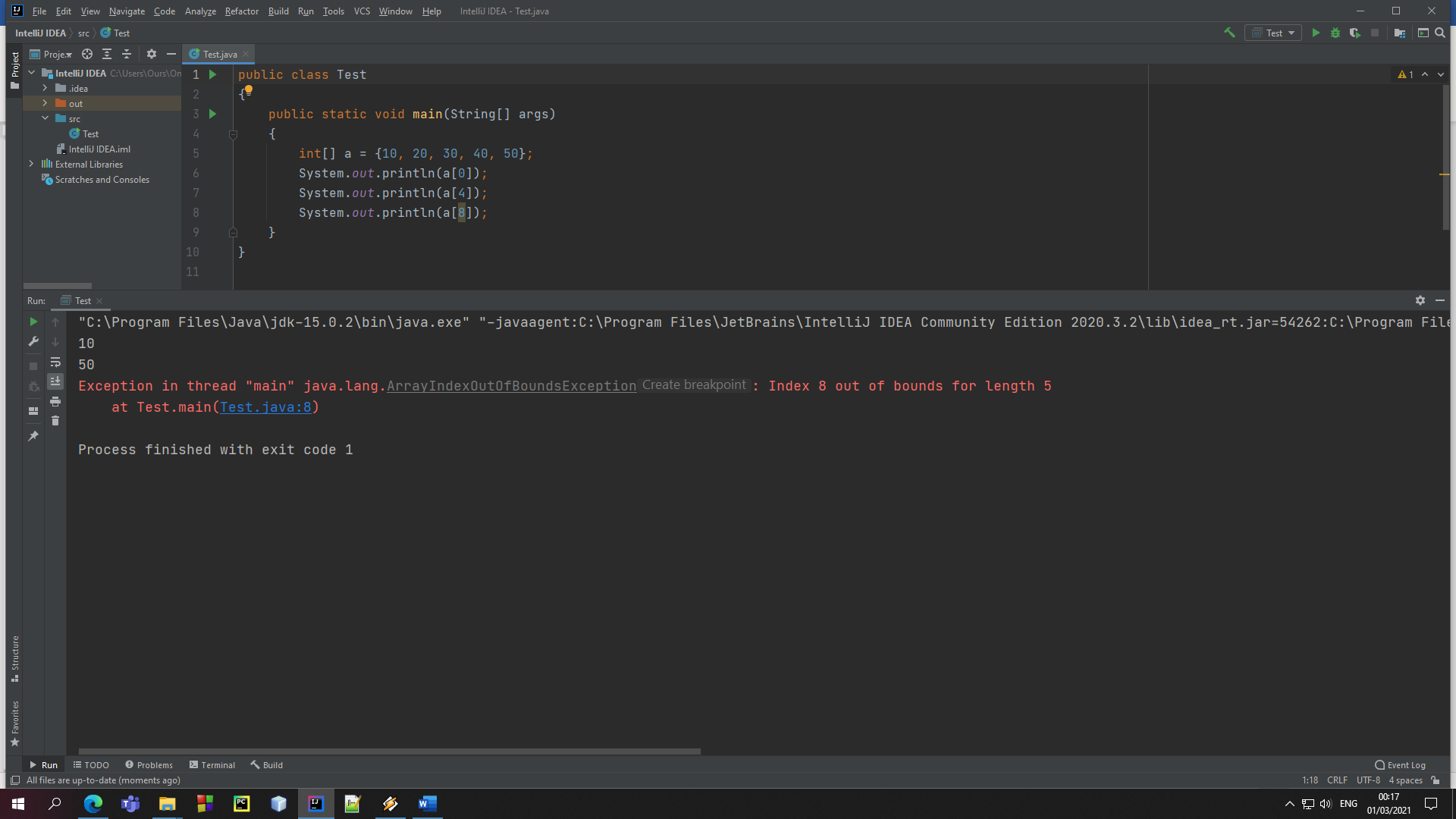
**int** b[] = **new int**[7];

System.out.println(b.length); **//se va afișa valoarea 7**

**for**(**int** elem : b) **//se vor afișa valorile 0 0 0 0 0 0 0**

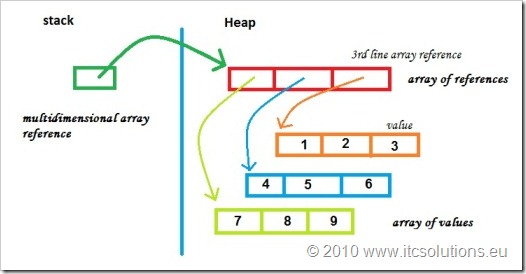
System.out.print(elem + " ");

Accesarea unui element dintr-un tablou al cărui indice este invalid (i.e., nu este cuprins între 0 și tablou.length-1) va duce la lansarea excepției ArrayIndexOutOfBoundsException în momentul rulării programului respectiv:



Observați faptul că excepția ArrayIndexOutOfBoundsException a fost lansată abia în momentul în care s-a încercat accesarea elementului a[8], valorile elementelor a[0] și a[4] fiind afișate corect!

În limbajul Java tablourile bidimensionale sunt, de fapt, tablouri unidimensionale ale căror elemente sunt referințe spre tablouri unidimensionale, fiecare reprezentând câte o linie (sursa imaginii: [Tutorial Java 6 - #4.3 Matrixes and Multidimensional Arrays | IT&C Solutions](https://www.itcsolutions.eu/2010/12/30/tutorial-java-6-4-3-matrixes-and-multidimensional-arrays/)):



Declararea tablourilor bidimensionale (de fapt, a unor referințe spre tablouri bidimensionale!) se poate realiza în mai multe moduri:

1. **tip\_de\_date[][] tablou\_1, tablou\_2, ..., tablou\_n;**
2. **tip\_de\_date tablou\_1[][], tablou\_2[][], ..., tablou\_n[][];**
3. **tip\_de\_date[] tablou\_1[], tablou\_2[], ..., tablou\_n[];**

Tablourile bidimensionale pot fi inițializate în momentul declarării, linie cu linie, așa cum se poate observa în exemplele de mai jos:

**int**[][] a = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

**int** a[][] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

**int**[] a[] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

Liniile unui tablou bidimensional pot să aibă lungimi diferite:

**int**[][] a = {{1, 2}, {3}, {4, 5, 6, 7}, {8, 9}};

**int** a[][] = {{1, 2, 3, 4, 5, 6}, {7, 8, 9}};

Alocare dinamică a unui tablou bidimensional având liniile de aceeași lungime se realizează astfel:

**referință\_tablou = new tip\_de\_date[numărReferinte][numărElemente];**

**Exemple:**

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

**int** a[][] = **new int**[7][7];

Alocare dinamică a unui tablou bidimensional având liniile de lungimi diferite se realizează astfel:

* se alocă un tablou bidimensional precizând doar numărul de linii:

**tablou = new tip\_de\_date[numărReferinte][];**

* se alocă, pe rând, fiecare linie din tabloul bidimensional, precizând numărul de coloane:

**tablou[0] = new tip\_de\_date[numărElemente];**

**tablou[1] = new tip\_de\_date[numărElemente];**

**...........................................**

**tablou[tablou.length-1] = new tip\_de\_date[numărElemente];**

**Exemplu:**

**int**[][] a = **new int**[3][];

a[0] = **new int**[7];

a[1] = **new int**[2];

a[2] = **new int**[5];

Parcurgerea unui tablou se poate realiza fie pozițional (prin intermediul indicilor elementelor), fie folosind o instrucțiune repetitivă de tipul **enhanced-for**.

**Exemplu:**

|  |  |
| --- | --- |
| **int**[][] a = {{1, 2}, {3}, {4, 5, 6, 7}, {8, 9}}; | |
| **for**(**int** i = 0; i < a.length; i++)  {  **for**(**int** j = 0; j < a[i].length; j++)  System.out.print(a[i][j] + " ");  System.out.println();  } | **// se va afișa (de două ori):**  **// 1 2**  **// 3**  **// 4 5 6 7**  **// 8 9** |
| **for**(**int**[] linie : a)  {  **for**(**int** elem : linie)  System.out.print(elem + " ");  System.out.println();  } |  |

**CLASA Arrays**

Clasa Arrays este o clasă utilitară (i.e., conține doar metode statice) dedicată manipulării tablourilor. În continuare, vom prezenta mai multe metode ale acestei clase, din versiunea sa existentă începând cu Java 11:

* **static String toString​(Tip[] tablou)** – furnizează o reprezentare a tabloului transmis ca parametru sub forma unui șir de caractere sau șirul "null" dacă referința sa este null. Elementele tabloului vor fi enumerate între o pereche de paranteze drepte și despărțite între ele prin câte o virgulă și un spațiu.

**Exemple:**

int[] t = {1, 2, 3, 4, 5};

int[][] a = {{1, 2}, {3}, {4, 5, 6}};

System.out.print(t); //[I@4dd8dc3

System.out.print(Arrays.toString(t)); //[1, 2, 3, 4, 5]

System.out.print(Arrays.toString(a)); //[[I@6d03e736, [I@568db2f2, [I@378bf509]

Observați faptul că pentru tablourile bidimensionale se vor furniza referințele tablourilor corespunzătoare liniilor sale!

* **static String deepToString​(Tip[] tablou)** – furnizează o reprezentare în adâncime a tabloului transmis ca parametru, respectiv dacă elementele tabloului sunt alte tablouri, atunci și acestea vor fi convertite în formatul precizat mai sus.

**Exemple:**

int[] t = {1, 2, 3, 4, 5};

int[][] a = {{1, 2}, {3}, {4, 5, 6}};

System.out.print(a); // [[I@4dd8dc3

System.out.print(Arrays.toString(a)); //[[I@6d03e736, [I@568db2f2, [I@378bf509]

System.out.print(Arrays.deepToString(a)); //[[1, 2], [3], [4, 5, 6]]

* **static void fill​(Tip[] tablou, Tip valoare)** – atribuie tuturor elementelor tabloului unidimensional valoarea indicată.

**Exemplu:**

int[] t = {1, 2, 3, 4, 5};

Arrays.fill(t, 7);

System.out.println(Arrays.toString(t)); //[7, 7, 7, 7, 7]

* **static boolean equals(Tip[] a, Tip[] b)** – verifică dacă tablourile a și b sunt egale, respectiv dacă au același număr de elemente și elementele aflate pe aceleași poziții sunt egale.

**Exemple:**

int[] t = {1, 2, 3, 4, 5};

int[] v = {1, 2, 3, 4, 5};

int[] w = {1, 2, 4, 4, 5};

System.out.println(Arrays.equals(t, v)); //true

System.out.println(Arrays.equals(w, v)); //false

int[][] a = {{1, 2}, {3}, {4, 5, 6}};

int[][] b = {{1, 2}, {3}, {4, 5, 6}};

System.out.println(Arrays.equals(a, b)); //false

Observați faptul că în cazul tablourilor bidimensionale a și b se va afișa false, deoarece se vor compara referințele liniilor lor!

* **static boolean deepEquals(Tip[] a, Tip[] b)** – verifică în profunzime dacă tablourile a și b sunt egale, respectiv dacă elementele lor sunt tot tablouri le verifică egalitatea din punct de vedere al lungimilor și al elementelor aflate pe același poziții, ci nu al referințelor.

**Exemplu:**

int[][] a = {{1, 2}, {3}, {4, 5, 6}};

int[][] b = {{1, 2}, {3}, {4, 5, 6}};

System.out.println(Arrays.deepEquals(a, b)); //true

* **static int mismatch(Tip[] a, Tip[] b)** – returnează cel mai mic indice k pentru care a[k] ≠ b[k] sau -1 dacă tablourile sunt egale. Dacă tablourile a și b nu sunt egale, atunci valoarea indicelui k este cuprinsă între 0 și minimul dintre lungimile celor două tablouri a și b.

**Exemple:**

int[] t = {1, 2, 3};

int[] u = {1, 2, 3};

int[] v = {1, 2, 3, 4, 5};

int[] w = {1, 2, 4, 5, 5};

System.out.println(Arrays.mismatch(t, u)); //-1

System.out.println(Arrays.mismatch(u, v)); //3

System.out.println(Arrays.mismatch(w, v)); //2

Observați faptul că Arrays.mismatch(u, v) returnează 3, deoarece u[0]==v[0], u[1]==v[1] și u[2]==v[2], dar în tabloul v există, în plus față de tabloul u, elementele 4 și 5, deci indicele 3 se referă strict la tabloul v!

* **static int compare(Tip[] a, Tip[] b)** – compară lexicografic cele două tablouri și returnează următoarele valori:
  + o valoare negativă dacă tabloul a este mai mic în sens lexicografic decât tabloul b;
  + o valoare pozitivă dacă tabloul a este mai mare în sens lexicografic decât tabloul b;
  + valoarea 0 dacă tabloul a este egal în sens lexicografic cu tabloul b.

**Exemple:**

int[] t = {1, 2, 3, 4, 5};

int[] u = {1, 2, 3};

int[] v = {1, 2, 3, 4, 5};

int[] w = {1, 2, 4, 4, 5};

System.out.println(Arrays.compare(t, v)); // 0

System.out.println(Arrays.compare(t, u)); // > 0

System.out.println(Arrays.compare(w, v)); // > 0, deoarece 4 = w[3] > v[3] = 3

System.out.println(Arrays.compare(v, w)); // < 0, deoarece 3 = v[3] < w[3] = 4

* **static Tip[] copyOf(Tip[] tablou, int nr\_elem)** – returnează un tablou format din primele nr\_elem elemente ale tabloului dat ca parametru. Dacă nr\_elem este strict mai mare decât lungimea tabloului, atunci se vor adăuga elemente nule de tip.

**Exemple:**

int[] t = {1, 2, 3, 4, 5};

int[] u = Arrays.copyOf(t, t.length);

int[] v = Arrays.copyOf(t, 3);

int[] w = Arrays.copyOf(t, 8);

System.out.println(Arrays.toString(u)); // [1, 2, 3, 4, 5]

System.out.println(Arrays.toString(v)); // [1, 2, 3]

System.out.println(Arrays.toString(w)); // [1, 2, 3, 4, 5, 0, 0, 0]

* **static void sort(Tip[] tablou)** – sortează crescător elementele tabloului dat ca parametru.

**Exemplu:**

int[] t = {2, 1, 5, 2, 5, 3, -10, 7, 4, 5};

Arrays.sort(t);

System.out.println(Arrays.toString(t)); // [-10, 1, 2, 2, 3, 4, 5, 5, 5, 7]

* **static int binarySearch(Tip[] tablou, Tip valoare)** – caută valoarea indicată în tabloul dat folosind algoritmul de căutare binară. Tabloul trebuie să fie sortat crescător, deoarece, în caz contrar, rezultatul furnizat nu va fi corect! Metoda returnează indicele pe care se găsește valoarea respectivă în tablou (dacă valoarea se găsește de mai multe ori în tablou, atunci se returna unul dintre indicii corespunzători) sau, dacă valoarea nu există în tablou, o valoare negativă p cu proprietatea că -p-1 reprezintă indicele unde ar putea fi inserată valoarea căutată în tablou astfel încât acesta să rămână sortat crescător (i.e., indicele unde ar fi trebuit să se găsească valoarea căutată).

**Exemple:**

int[] t = {1, 2, 7, 7, 10, 10, 10, 21};

System.out.println(Arrays.binarySearch(t, 10)); // 5

System.out.println(Arrays.binarySearch(t, 9)); // -5, deoarece valoarea 9 ar

// trebui să se găsească în tablou pe poziția -(-5)-1 = 4

System.out.println(Arrays.binarySearch(t, -9)); // -1, deoarece valoarea -19 ar

// trebui să se găsească în tablou pe poziția -(-1)-1 = 0

System.out.println(Arrays.binarySearch(t, 99)); // -9, deoarece valoarea 99 ar

// trebui să se găsească în tablou pe poziția -(-9)-1 = 8

În afara metodelor prezentate mai sus, în clasa Arrays mai sunt definite și alte metode, majoritatea fiind variante ale celor prezentate mai sus (de exemplu, metode care nu prelucrează un tablou întreg, ci doar o secvență a sa, precizată prin 2 indici): <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Arrays.html>.

**CLASE ȘI OBIECTE**

În continuare, vom prezenta maniera de implementare a principiilor *programării orientate pe obiecte* (POO) în Java. Paradigma POO în sine este evident aceeași cu cea pe care deja ați studiat-o în C++. Diferențele apar din punct de vedere al modului de implementare.

* **Principiile de bază ale POO:**

1. **Abstractizarea (abstraction)**

* Pe lângătipurile de date predefinite în limbajul Java (primitive sau clase), cunoscute de mașina virtuală, în diferite aplicații este nevoie de noi tipuri de date, care să modeleze un concept sau un fenomen din lumea reală. Modelarea se realizează prin abstractizare, adică se identifică acele caracteristici/atribute relevante în contextul aplicației respective. Pe lângă caracteristicile identificate, se stabilește și un set de operații (comportament) care să acționeze asupra datelor respective.

1. **Încapsularea (encapsulation)**

* Încapsularea reprezintă mecanismul prin care datele și operațiile specifice sunt înglobate sub forma unui tot unitar – un obiect.
* Datele obiectului sunt ascunse (principiu ascunderii) pentru a nu putea fi accesate incorect în anumite prelucrări. Modificarea și accesarea lor se poate realiza prin intermediul unor metode publice de tip set/get.
* De asemenea, prin încapsulare se separa detaliile de implementare față de implementarea propriu-zisă. Un utilizator trebuie doar să acceseze anumite operații, ci nu să cunoască detalii complexe de implementare.

1. **Moștenirea (inheritance)**

* Proprietatea prin care o clasă preia date și metode dintr-o clasă definită anterior, în scopul reutilizării codului

1. **Polimorfismul (polymorphism)**

* Proprietatea unui obiect de a avea comportament diferit, în funcție de context.
* **Supraîncărcare (overloading/polimorfism static)** = pot exista două sau mai multe metode cu același nume, dar având listele parametrilor diferite.
* **Suprascriere (overriding)** =o subclasă rescrie o metodă a superclasei.
* **Clasa** este o implementare a unui tip de date abstract și poate fi privită ca un șablon pentru o categorie de obiecte.
* **Sintaxa unei clase:**

[modificatori] **class** denumireClasă {

date membre/atribute

metode membre **//nu mai pot fi implementate în afara clasei!**

}

* + - * **Modificatorii de clasă:**
* **public**: clasa poate fi accesată/instanțiată și din afara pachetului său
* **abstract:** clasa conține cel puțin o metodă fără implementare (metodă abstractă) și nu poate fi instanțiată
* **final:** clasa nu mai poate fi extinsă

*Observație*: Dacă nu exista modificatorul public, clasa are un acces implicit, adică poate fi accesată/instanțiată doar din interiorul pachetului în care a fost creată.

În general, o clasă conține date membre, constructori și metode.

* **Date membre**
* Datele membre pot fi de orice tip (primitiv sau referință).
* Se declară ca orice variabilă locală, însă declararea poate fi însoțită și de modificatori.
* Datele membre sunt inițializate cu valori nule de tip (spre deosebire de variabilele locale)!
* **Modificatori pentru date membre:**

1. **Modificatorii de acces:**
   * **public**: data membră poate fi accesată și din afara clasei, însă în conformitate cu principiul ascunderii (încapsulare) acestea sunt, de obicei, private
   * **protected**: data membră poate fi accesată din clasele din același pachet sau de subclasele din ierarhia sa
   * **private**: data membră poate fi accesată doar din clasa din care face parte
   * dacă nu este precizat niciun modificator de acces, atunci data membră respectivă are acces implicit, adică poate fi accesată doar din sursele aflate în același pachet
2. **Alți modificatori:** 
   * **static:** data membră este un câmp de clasă, adică este alocat o singură dată în memorie și partajat de toate instanțele clasei respective
   * **final:** data membră poate fi doar inițializată, fără a mai putea fi modificată ulterior. Dacă data membră este un obiect, atunci nu i se poate modifica referința, dar conținutul său poate fi modificat!

* **Pentru o dată membră se pot combina mai mulți modificatori!**
* În concluzie, datele membre se împart doua categorii:
* **Date membre de instanță = date membre non-statice:** se multiplică pentru fiecare obiect, alocându-se spațiu de memorie pentru fiecare în parte și sunt inițializate prin constructori.
* **Date membre de clasă = date membre statice:** sunt partajate de către toate obiectele, se alocă o singură dată și pot fi modificate de orice instanță (obiect) al clasei respective**.** Exemplu: definirea unei date membre care nu depinde de un anumit obiect (TVA) sau pentru a contoriza numărul de obiecte instanțiate. Datele membre statice nu trebuie inițializate prin constructori!!!

**Exemplu:**

**class** Persoana{

**private int** IDPersoana;

**private** String nume;

**private int** varsta;

**private static String** nationalitate = "română";

**private static int** nrPersoane = 0;

.........

}

* **Metode**
  + - * **Sintaxa unei metode:**

[modificatori] tipReturnat numeMetoda ([parametri]){

//corpul metodei

}

* + - Modificatorii sunt aceiași ca la date membre, dar se adaugă și modificatorul **abstract** prin care se declară o metodă fără implementare, care urmează să fie definită obligatoriu în subclasele clasei respective.
    - Utilizarea modificatorului **final** pentru o metodă împiedică redefinirea sa în subclasele clasei respective. De exemplu, o metodă care calculează TVA conține o formulă de calcul unică, care nu trebuie modificată/particularizată de către subclasele sale.
    - Parametrii unei metode nu pot să aibă valori implicite.
    - **Parametrii unei metode sunt transmiși întotdeauna doar prin valoare!**

**Exemplu:**

Considerăm următoarea clasă:

**public class** Test {

**static void** modificare(**int** v[]) {

v[0] = 100;

v = **new int**[10];

v[1] = 1000;

}

**public static void** main(String[] args) {

**int** v[] = {1, 2, 3, 4, 5};

modificare(v);

System.out.println(Arrays.toString(v));

}

}

După rulare, se va afișa următorul tablou: [100, 2, 3, 4, 5].

* + - Metodele statice nu pot accesa date membre sau metode non-statice, dar invers este posibil.
    - Într-o clasă pot exista mai multe metode cu același nume prin intermediul mecanismului de supraîncărcare (**overloading**).

**class** Persoana {

.........

**public** String getNume() {

**return** nume;

}

**public void** setNume(String nume) {

**this**.nume = nume;

}

**public static void** afisareNumarPersoane(){

System.out.println("Numar persoane: " + nrPersoane);

}

}

* + - Două metode cu același nume se consideră ca fiind supraîncărcate dacă diferă prin numărul sau tipul parametrilor lor.
    - Dacă două metode au același nume și aceeași listă a parametrilor, dar diferă prin tipul returnat, atunci ele nu se vor considera supraîncărcate și compilatorul va semnaliza o eroare.
* **Referința this**
* Referința **this** reprezintă referința obiectului curent, respectiv a obiectului pentru care se accesează o dată membru sau o metodă.
* Modalități de utilizare:
* pentru a accesa o dată membră sau pentru a apela o metodă:

**this**.nume="Popa Ion"

**this**.afișarePersoană();

* pentru a diferenția într-o metodă o dată membru de un parametru cu aceleași nume:

**public void** setNume(String nume) {

**this**.nume = nume;

}

* **Constructori**
  + - Constructorii au rolul de a inițializa datele membre.
    - Un constructor are numele identic cu cel al clasei și nu returnează nici o valoare.
    - Un constructor nu poate fi static, final sau abstract.
    - O clasă poate să conțină mai mulți constructori, prin mecanismul de supraîncărcare.
    - Dacă într-o clasă nu este definit niciun constructor, atunci compilatorul va genera unul implicit (default), care va inițializa toate datele membre cu valorile nule de tip, mai puțin pe cele inițializate explicit!
    - **Tipuri de constructori:**
* **cu parametri:** inițializează datele membre cu valorileparametrilor

**public** Persoana(String nume, **int** varsta) {

**this.nume = nume;**

**this.varsta = varsta;**

}

* **fără parametri:** inițializează datele membre cu valori constante

**public** Persoana() {

**this**.nume = "Popa Ion";

**this**.varsta = 20;

}

* + - Pentru a apela constructorul cu argumente se poate utiliza referința **this**:

**public** Persoana() {

**this(**"Popa Ion",20);

}

* + - De obicei, un constructor este public, dar poate fi și privat într-unul din următoarele cazuri:
* nu dorim să fie instanțiată o anumită clasă (de exemplu, dacă aceasta este o clasă de tip utilitar care conține doar date membre/metode statice - clasa Math)
* dorim să instanțiem un singur obiect din clasa respectivă (clasă singleton)
* **Exemplu pentru Singleton Design Pattern:**
  + - Considerăm o aplicație Java care modelează activitatea dintr-o organizație utilizând câte o clasă pentru fiecare rol (angajat, director de departament, președinte) în parte. Evident, orice organizație are un singur președinte, deci clasa President care modelează acest rol trebuie să permită o singură instanțiere a sa!
    - Pentru a realiza o instanțiere unică a clasei President vom proceda astfel:
* constructorul implicit va fi privat, pentru a împiedica instanțierea clasei;
* vom utiliza un câmp static care pentru a reține referința singurei instanțe a clasei;
* vom utiliza o metodă statică de tip *factory* pentru a furniza referința spre singura instanță a clasei.

**class** President {

**private static** String name;

**private static** President president;

**private** President() {

President.name = "Mr. John Smith";

}

**public static** President getPresident() {

**if** (president == null)

president = **new** President();

**return** president;

}

**public static void** showPresident(){

System.out.println("President: " + name);

}

}

**public class** Test{

**public static void** main(String[] args) {

President p = President.getPresident();

President q = President.getPresident();

System.out.println(p == q);

}

}

* + - Se observă faptul că referințele p și q sunt egale, deci am realizat o clasă singleton!
    - De asemenea, se observă faptul că singura instanță a clasei este creată doar în momentul în care aceasta este solicitată, adică este apelată metoda factory getPresident. În acest caz spunem că se realizează o *instanțiere târzie* (lazy initialization).
    - O altă variantă de implementare a unei clase singleton constă în crearea singurei instanțe a sa chiar din momentul în care clasa este încărcată de mașina virtuală Java, printr-o *inițializare timpurie* (early initialization):

**class** President {

**private static** String name;

**private static final** President president = **new** President();

**private** President() {

name = "Mr. John Smith";

}

**public static** President getPresident() {

**return** president;

}

**public void** showPresident(){

System.out.println("President: " + name);

}

}

* **Constructorul de copiere**

**În limbajul Java nu există constructor de copiere având funcționalitățile din limbajul C++!** Evident, o clasă poată să conțină un constructor având ca parametru un obiect al clasei respective, în scopul de a copia în obiectul curent datele membre ale obiectului transmis ca parametru. Totuși, acest constructor **nu va fi apelat automat** în cazurile în care se apelează un constructor de copiere în alte limbaje orientate obiect (de exemplu, în limbajul C++, pentru a realiza o copie a unui parametru al unei metode transmis prin valoare).

În limbajul Java datele membre pot fi copiate în două moduri:

* **bitwise/shallow copy**: se copiază datele membre bit cu bit, inclusiv referințele!

**Exemplu:**

Adăugăm în clasa Persoana data membră double[] venit care va conține veniturile persoanei respective în fiecare dintre cele douăsprezece luni ale unui an:

**class** Persoana {

**private int** IDPersoana;

**private** String nume;

**private int** varsta;

**private static** String nationalitate = "română";

**private static int** nrPersoane = 0;

**private double** []venit;

**public** Persoana() {

**this**.nume = "";

**this**.venit = **new double**[12];

}

**public** Persoana(String nume, **int** varsta, **int**[] venit) {

**this**.nume = nume;

**this**.varsta = varsta;

**this.**venit = venit;

}

**public** Persoana(Persoana ob){

**this**(ob.nume, ob.varsta, ob.venit);

}

**public void** setVenit(**int** luna, **double** suma){

venit[luna] = suma;

}

**public double** getVenit(**int** luna){

**return** venit[luna];

}

...............

}

Considerăm acum următoarea secvență de cod:

Persoana p1 = **new** Persoana("Pop Ana", 24,

**new double**[]{2000.25,3000.50,4000.75});

Persoana p2 = **new** Persoana(p1);

p2.setVenit(2, 5000);

System.out.println("Veniturile persoanei p1: " + p1.getVenit(2));

System.out.println("Veniturile persoanei p2: " + p2.getVenit(2));

După rularea secvenței de cod de mai sus, se va afișa:

Veniturile persoanei p1: 5000.0

Veniturile persoanei p2: 5000.0

Din cauza faptului că prin utilizarea constructorului Persoana(Persoana ob) în obiectul p2 a fost copiată referința tabloului venit din obiectul p1, modificarea venitului persoanei p2 într-o anumită lună a condus și la modificarea venitului persoanei p1 în luna respectivă!

* **deep copy:** pentru datele membre de tip referință se alocă mai întâi spațiu de memorie, după care se copiază explicit conținutul acestora. Fara imutabile

**Exemplu:**

Modificăm în clasa Persoana constructorul Persoana(Persoana ob)astfel:

**public** Persoana(Persoana ob) {

**this**.nume = ob.nume;

**this**.varsta = ob.varsta;

**this.venit = Arrays.copyOf(ob.venit, ob.venit.length);**

}

* **Blocuri de inițializare**
* Constructorii nu trebuie să inițializeze date membre statice, dar le pot manipula.

**Exemplu:** pentru a inițializa data membră IDPersoana cu numărul curent de obiecte de tip Persoana instanțiate, trebuie să adăugăm în fiecare constructor instrucțiunea this.IDPersoana = ++nrPersoane.

* Totuși, există posibilitatea de a folosi un **bloc static de inițializare** a datelor membre statice. Acest bloc se apelează o singură dată, când JVM încarcă clasa respectivă. De obicei, un astfel de bloc se folosește când sunt necesare inițializări mai complexe ale datelor membre statice (conectarea la o bază de date sau un server, citirea unor informații dintr-un fișier etc.).

**Exemplu:** eliminăm din clasa Persoana inițializările directe ale datelor membre statice și adăugăm un bloc static de inițializare:

**class** Persoana {

...............

**private static** String nationalitate;

**private static int** nrPersoane;

**static**{

nationalitate = "română";

nrPersoane = 0;

}

...............

}

* De asemenea, se poate declara un **bloc nestatic de inițializare**, care va fi executat înaintea fiecărui apel de constructor. Acest bloc conține, de regulă, o secțiune de cod comună tuturor constructorilor. De exemplu, pentru a inițializa data membră IDPersoana cu numărul curent de obiecte de tip Persoana instanțiate, nu mai adăugăm instrucțiunea this.IDPersoana = ++nrPersoane în fiecare constructor, ci folosim un bloc nestatic de inițializare:

**class** Persoana {

**private int** IDPersoana;

**private static int** nrPersoane;

...............

**static**{

nationalitate = "română";

nrPersoane = 0;

}

{

**this**.IDPersoana = ++nrPersoane;

}

**public** Persoana() {

**this**.nume = "";

**this**.venit = **new double**[12];

}

...............

}

**Exemplu:**

class Persoana {

String nume;

int varsta, ID;

static int nrPersoane;

static String nationalitate;

public Persoana() {

this.nume = "XYZ";

this.varsta = 20;

System.out.println("Constructorul cu 0 parametri");

}

public Persoana(String nume) {

this.nume = nume;

this.varsta = 20;

System.out.println("Constructorul cu 1 parametru");

}

public Persoana(String nume, int varsta) {

this.nume = nume;

this.varsta = varsta;

System.out.println("Constructorul cu 2 parametri");

}

//bloc nestatic de initializare

{

this.ID = ++nrPersoane;

System.out.println("Bloc nestatic de initializare");

}

//bloc static de initializare

static

{

System.out.println("Bloc static de initializare\n");

nationalitate = "română";

nrPersoane = 0;

}

@Override

public String toString() {

return "nume='" + nume + ", varsta=" + varsta + ", ID=" + ID;

}

}

public class Test

{

public static void main(String[] args)

{

Persoana p1 = new Persoana();

System.out.println(p1);

System.out.println();

Persoana p2 = new Persoana("Popescu Ion");

System.out.println(p2);

System.out.println();

Persoana p3 = new Persoana("Ionescu Dana", 25);

System.out.println(p3);

System.out.println();

}

}

* **Obiecte** 
  + **Ciclul de viață al unui obiect:** 
    - declararea unei referințe (și inițializarea sa cu null dacă este o variabilă locală):

Persoana p = null;

* + - instanțierea obiectului folosind operatorul new (alocare dinamică în heap) și un constructor al clasei respective:

p = **new** Persoana(nume, vârsta);

* + - un obiect poate fi instanțiat și în momentul declarării sale:

Persoana p = **new** Persoana(nume, vârsta);

* + - utilizarea obiectului prin intermediul metodelor sale publice:

p.setNume("Popescu Ion");

System.out.println(p.getNume());

* + - distrugerea automată a obiectului
  + O data membră/metodă statică poate fi apelată și cu o referință null:

Persoana p = null;

p.afisareNumarPersoane()

* + **Distrugerea obiectelor se realizează automat în limbajul Java!**
    - **Garbage Collection** este mecanismul prin care JVM eliberează spațiul alocat unor obiecte care nu mai sunt folosite, utilizând un fir de executare dedicat, numit **Garbage Collector (GC)**. Acest fir scanează memoria și verifică faptul că o zonă de memorie mai este utilizată sau nu, marcând zonele nefolosite. Ulterior, zonele de memorie marcate sunt eliberate (sunt raportate ca fiind libere) și, eventual, se realizează o compactare a memoriei.
    - Un obiect se consideră ca fiind neutilizat dacă, de exemplu:
      * nu mai există nicio referință, directă sau indirectă, spre obiectul respectiv;
      * obiectul a fost creat în interiorul unui bloc (local) și executarea blocului respectiv s-a încheiat;
      * obiectul face parte dintr-o insulă de izolare (*island of isolation*), adică un grup de obiecte între care există referințe, dar spre niciunul dintre ele nu mai există referințe din exteriorul grupului (<https://javasolutionsguide.blogspot.com/2015/08/how-to-make-object-eligible-for-garbage.html>).
    - Înainte de a distruge un obiect, GC apelează metoda finalize pentru a-i oferi obiectului respectiv posibilitatea de a mai executa un set de acțiuni.
    - Un obiect neutilizat nu va fi neapărat distrus imediat și nu se poate forța pornirea GC folosind System.gc() sau Runtime.getRuntime().gc()!