METODE AVANSATE DE PROGRAMARE

Conf.univ.dr. Ana Cristina DĂSCĂLESCU

Tematică curs 2

Tablouri

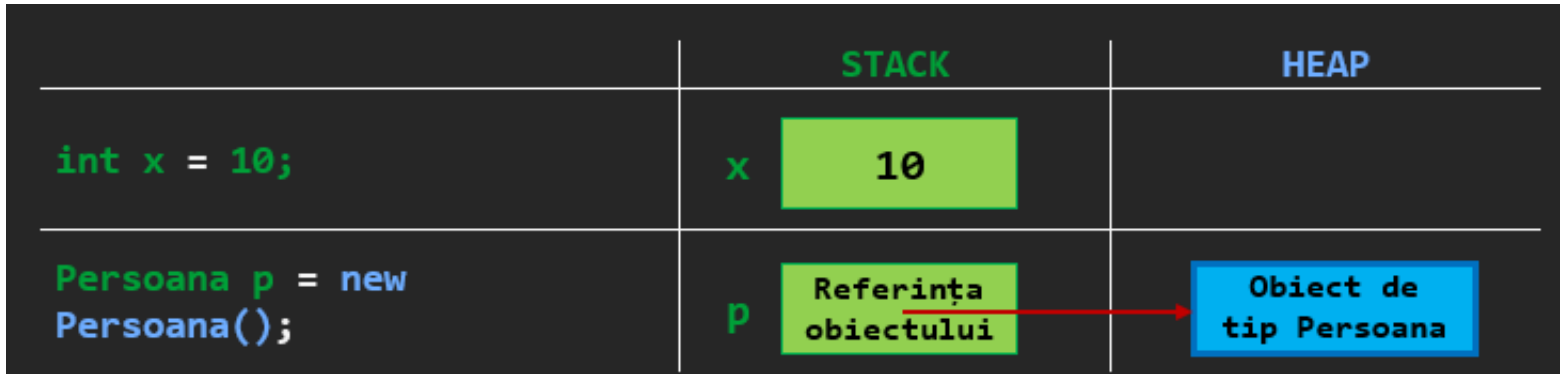
- Declarare, instanțiere
- Clasa java.util.Arrays

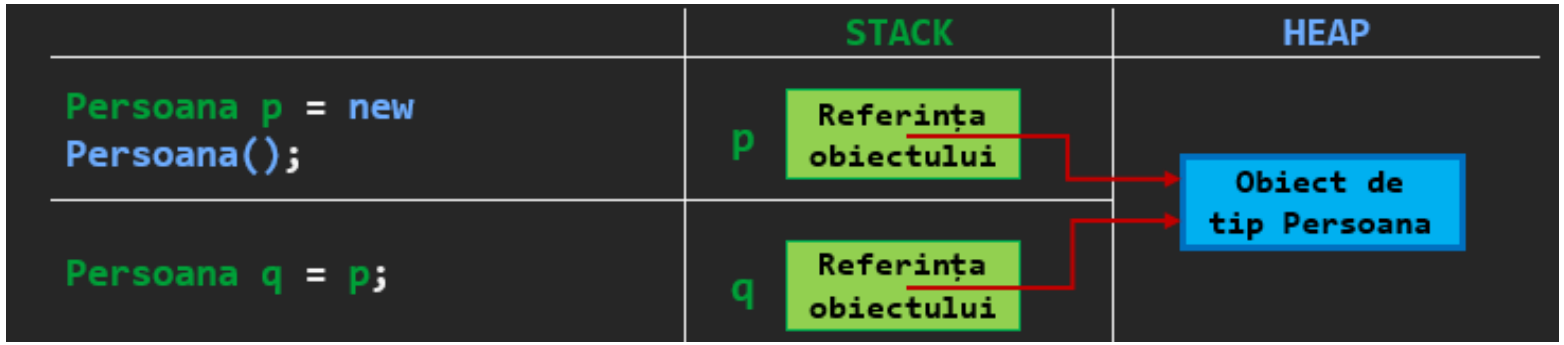
Clase și obiecte

- Definirea unei clase
- Membrii unei clase
- Tipuri de constructori
- Ciclul de viață al unui obiect
- Extinderea claselor

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.





Tipuri de date de referință

- În Java tipuri de date de referință sunt:

✓Tablouri

✓Clase

✓ Interfeţe

✓Enumerări

Tablouri (Arrays)

➢ **Tablou** este un obiect de tip container care conține elemente de același tip.

- Tabloul este o referință -> trebuie sa fie inițializat sau alocat dinamic înainte de a fi utilizat

➢ **Declararea unui tablou unidimensional**

```
tipData numeTablou[];
tipData []numeTablou;
```

- La declararea unui tablou se crează doar o referință, alocată în zona de memorie stack!

Tablouri (Arrays)

- **Exemple**

```
int v[], w;
```

v este o referință null, w este o variabilă de tip întreg

```
int []v, w;
```

v și w sunt referințe null

```
String [] args;
```

Tablouri (Arrays)

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

a) `tip_de_date[] tablou = {valoare_1, ..., valoare_n};`

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

a) `tip_de_date tablou[] = {valoare_1, ..., valoare_n};`

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

Tablouri (Arrays)

- 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];

Tablouri (Arrays)

➢Observații

- Dimensiunea stabilită la instanțiere nu mai poate fi modificată.

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

- Pentru orice obiect de tip tablou este definit un câmp **length** public, întreg, static și final (constant) care memorează dimensiunea tabloului instanțiat:

```
int []a = new int[10];
for(int i=0; I < v.length; i++)
    v[i] = i+1;
```

Tablouri (Arrays)

- Elementele unui tablou pot fi inițializate la instanțiere:

```
String []colors = {"Red", "Green", "Blue"};
int v[] = {1, 3, 5, 7, 9};
```

- Pentru orice obiect de tip tablou este definit un câmp **length** public, întreg, static și final (constant) care memorează dimensiunea tabloului instanțiat:

```
int []a = new int[10];
for(int i=0; I < v.length; i++)
    v[i] = i+1;
```

Tablouri (Arrays)

➢Observații

- Dimensiunea stabilită la instanțiere nu mai poate fi modificată.

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

- 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

```
int v[] = new int[3];
System.out.println(v[0]);
System.out.println(v[3]);
```

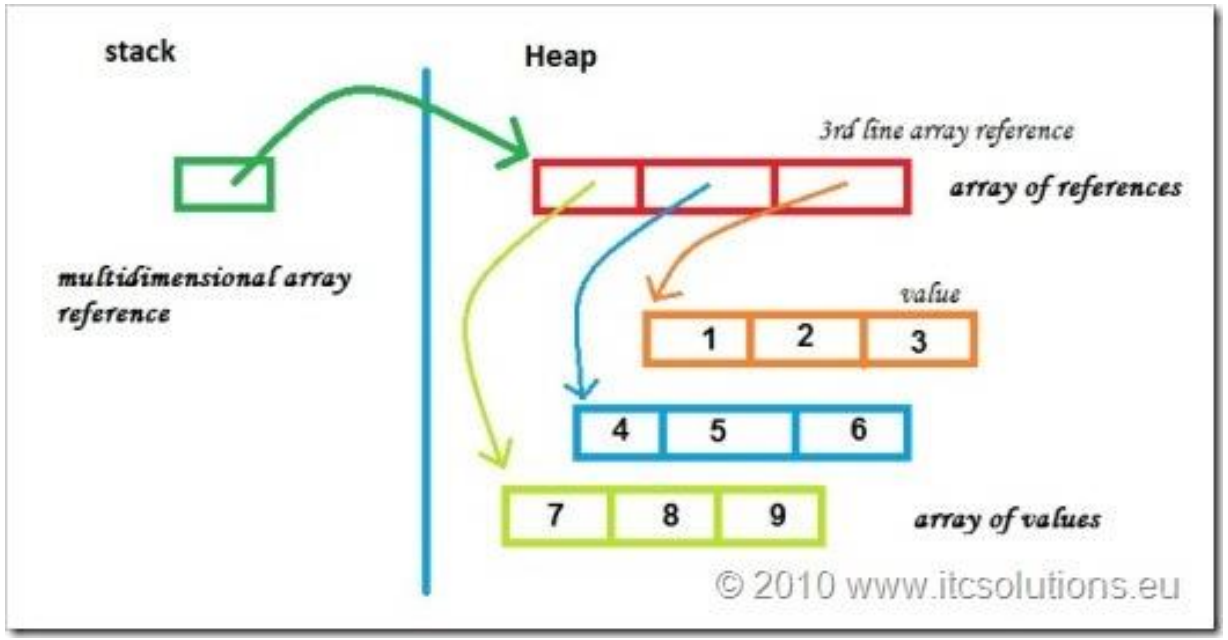
Se va afișa valoarea 0

Va fi generată excepția ArrayIndexOutOfBoundsException

Tablouri (Arrays)

➢Tablouri bidimensionale

- În limbajul Java tablourile bidimensionale sunt, de fapt, tablouri unidimensionale ale căror elemente sunt referințe spre tablouri unidimensionale.



Tablouri (Arrays)

➢ Tablouri bidimensionale

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

```
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}};
```

Tablouri (Arrays)

- 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];
```

›

- Se poate aloca 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];
```

Tablouri (Arrays)

➢ Clasa java.util.Arrays

- Este o clasă utilitară (conține doar metode statice și publice) care oferă implementări specifice unui tablou, respectiv sortare, căutare, copiere etc.
- `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.
- `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.
- `static void sort(Tip[] tablou)` – sortează crescător elementele tabloului dat ca parametru

Clase și obiecte

➢ Principii de bază ale programării orientate obiect

- Abstractizarea: permite crearea unor noi tipuri de date
- Încapsularea: reprezintă mecanismul prin care datele și operațiile specifice sunt înglobate sub forma unui tot unitar
- Moștenirea: proprietatea prin care o clasă preia date și metode dintr-o clasă definită anterior

› Polimorfismul: proprietatea unui obiect de a avea comportament diferit în funcție de context

Clase și obiecte

➢Clasa

- Este o implementare a unui **tip de date de referință** și poate fi privita 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!
}
```

Clase și obiecte

➢ Modificatorii de clasă

- public: clasa poate fi 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 instanțiată doar din interiorul pachetului în care a fost creată!

Clase și obiecte

➢ Date membre

- Datele membre pot fi de orice tip, respectiv 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)!

- Sintaxa:

```
[modificatori] tip dataMembra = [val init];
```

Clase și obiecte

➢ Modificatori pentru date membre:

- **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

- **Observație**: 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!

Clase și obiecte

- **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!

- **Observație**: Pentru o dată membră se pot combina mai mulți modificatori!

- Exemplu: `public static String facultate = "Informatica";`

Clase și obiecte

➢ Metode membre

- Oferă implementări concrete ale operațiilor care se execută asupra datelor membre.
- Setul de metode membre descrie funcționalitatea unui obiect.

- Sintaxa unei metode:

```
[modificatori] tipReturnat numeMetoda ([parametri]){
  //corpul metodei
}
```

- Modificatorii unei metode membre sunt similari cu cei specifici datlor membre, la care se adaugă și modificatorul **abstract** prin care se declară o metodă fără implementare.

➢Observații

- Utilizarea modificatorului final pentru o metodă membră împiedică redefinirea sa în subclasele clasei respective.
- Metodele statice nu pot accesa date membre sau metode non-statice.
- Într-o clasă pot exista mai multe metode cu același nume prin intermediul mecanismului de supraîncărcare (overloading).

Clase și obiecte

➢ Observații

- Parametrii unei metode sunt transmiși întotdeauna doar prin valoare!

- Exemplu:

```
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));
    }
}
```

Clase și obiecte

➢ Referința this

- Reprezintă referința obiectului curent, respectiv a obiectului pentru care se accesează o dată membru sau o metodă membră.

- Referința this se poate utiliza în următoarele cazuri:

✓ 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 denumire:

```
public void setNume(String nume) {
this.nume = nume;
}
```

Clase și obiecte

➢ 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!

Clase și obiecte

➢ Tipuri de constructori:

- **cu parametri:** inițializează datele membre cu valorile parametrilor

```
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;
}
```

Clase și obiecte

- De obicei, un constructor este public, însă există și situații în care acesta poate fi privat:

✓este necesar ca o clasă să nu fie instanțiată, de exemplu, dacă aceasta este o clasă de tip utilitar care conține doar date membre/metode statice.

Exemple: clasele `java. ava.lang.Math, java.util.Arrays`

✓este necesar ca o clasă sa aibă o singură instanță (clasă singleton)

- **Observație:** În limbajul Java nu există constructor de copiere!

Clase și obiecte

➢Clasa singleton

- Structura unei clase singleton:

✓constructorul clasei este privat, pentru a împiedica astfel instanțierea clasei;

✓clasa conține un câmp static care pentru a reține referința singurei instanțe a clasei;

✓clasa conține o metodă statică de tip *factory* pentru a furniza referința spre singura instanță a clasei.

Clase și obiecte

- Exemplu:

```
class President {
    private static String name; //câmp de instanță
    private static President president;

    private 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);
    }}
```

Clase și obiecte

➢ Ciclul de viață al unui obiect:

- **Declararea obiectului** presupune definirea unei variabile alocată în zona de memorie stivă care va reține adresa obiectului după ce acesta este instanțiat.

✓ Dacă declararea obiectului se realizează local, în cadrul unei metode, atunci inițializarea sa cu `null` este obligatorie.

```
void medtoda ()
{
  Persoana p = null;
  ..............
}
```

Clase și obiecte

- **Instanțierea obiectului**

✓ presupune alocarea unei zone de memorie HEAP.

✓ Alocarea zonei de memorie HEAP se realizează folosind operatorul **new** care returnează adresa de memorie alocată sau `null` dacă alocarea nu s-a realizat cu succes.

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

- **Funcționalitatea obiectului** este asigurată de setul metodelor publice.

```
p.setNume("Popescu Ion");
System.out.println(p.getNume());
```

- **Observație:** O data membră/metodă statică poate fi apelată și cu o referință `null`:

```
Persoana p = null;
p.afisareNumarPersoane()
```

Clase și obiecte

- **Eliberarea zonei de memorie**

✓ Se realizează automat de către mașina virtuală Java prin procesul **Garbage Collection** care conține un fir de executare dedicat, cu o prioritate scăzută, denumit **Garbage Collector (GC)**

✓ Un obiect devine eligibil pentru **Garbage Collector** în următoarele situații:

- 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
- dacă un obiect container conține o referință spre un alt obiect și obiectul container este devine `null`

✓ 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.