Lab.3: Clase în Java

Cuprins

1. Recapitulare laborator 2

În laboratorul 2, am introdus tablourile în Java, am văzut cum se declară şi cum se creează acestea, cum se poate copia un tablou în altul şi am descoperit metode din clasa `Arrays`, foarte folositoare atunci când se lucrează cu tablouri.

Am continuat cu aprofundarea clasei `Scanner`, folosită pentru citirea datelor de la tastelor de la tastatură sau, aşa cum vom vedea, dintr-un fişier, iar apoi am introdus noi metode ale clasei `String`.

Încercaţi să răspundeţi la următoarele întrebări:

- Cum se creează un tablou în Java ?
- Ce tip de date este un tablou ?
- Ce puteţi spune despre lungimea unui tablou ?
- Cum se realizează copierea unui tablou în altul ?
- Cum se poate testa că următoarea entitate de citit are tipul dorit, folosind clasa `Scanner`?
- Cum se poate verifica egalitatea stringurilor în Java?

2. Clase

Clasele reprezintă tipuri de date definite de utilizator sau deja existente în sistem. O clasă poate conţine:

- **câmpuri** (*variabile membru*, care definesc starea obiectului)
- **metode** (*funcţii membru*, ce reprezintă operaţii asupra stării)
- **constructori** (metode speciale, folosite la instanţierea clasei).

Prin instantierea unei clase se înţelege crearea unui obiect care corespunde tiparului definit de clasa respectivă. În cazul general, acest lucru se realizează prin intermediul cuvântului cheie `new`.

Exemplu: Dacă C este o clasă, atunci crearea unui obiect de tipul clasei C se poate face astfel:

- `new C(...);` // se creează un obiect *anonim* de tipul C;
- `C ob;` //declararea varabilei

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

  sau prescurtat:

`C ob = new C(...);` // se creează un obiect de tipul `C` la care ne putem referi prin variabila `ob`.

Observaţii:

- 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 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 parametrii, care nu prevede nici o acțiune. În acest caz, la creare trebuie folosită forma **`new`** `C()`.
- Variabila referință `ob` poate fi folosită pentru accesarea câmpurilor şi 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(...);` (evident, este invocată acea metodă care are aceeaşi signatură cu cea din lista de argumente).

Declararea unei clase:

[**public**][**abstract**][**final**] **class** <NumeClasă> [**extends**] <NumeSuperClasă> [**implements** Interfață1 [, Interfață2 ...]] {

```
    // Variabile membru

    // Constructori

    // Metodele clasei
}
```

Observaţii:

- O variabilă membru poate fi de un tip primitiv (numeric, caracter etc.) sau o referință.
- Dacă **nu** inițializăm valorile câmpurilor explicit, maşina virtuală va seta toate referințele la **null** şi tipurile primitive la 0 (pentru tipul boolean la **false**).
- O variabilă de tip referință cu valoarea null, 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, în caz contrar se va arunca excepția NullPointerException.


Implicit, o clasă poate fi folosită doar de clasele aflate în acelaşi pachet (librărie) cu clasa respectivă (dacă nu se specifică un anume pachet, toate clasele din directorul curent sunt

considerate a fi în acelaşi pachet). O clasă declarată public poate fi folosită din orice altă clasă din care este vizibilă (care are acces la pachetul din care face parte).

 O clasă poate fi definită şi ca protected sau private, însă aceste tipuri de clase nu pot fi de sine stătătoare, ci **este obligatoriu** să fie incluse într-o altă clasă. Ele se numesc **inner classes**, iar clasa care le cuprinde se numeşte **outer class**.

 **Exemplu:** Vom scrie o clasă Dreptunghi ce are două câmpuri, repezentând lungimea şi lățimea dreptunghiului şi doi constructori: un constructor fără parametrii în care se inițializează lungimea şi lățimea dreptunghiului cu 1 şi un constructor cu doi parametrii, reprezentând lungimea şi lățimea dreptunghiului. Clasa va avea următoarele metode:

- o metodă de calcul al ariei dreptunghiului
- o metodă ce primeşte ca parametru un obiect de tip Dreptunghi şi verifică dacă aria acestuia este mai mare decât aria dreptunghiului curent
- o metodă de afişare a dimensiunilor dreptunghiului

```
class Dreptunghi {
   double lung, lat; //campuri

   Dreptunghi() { //constructori
       lung = 1;
       lat = 1;
   }

   Dreptunghi(double lung1, double lat1) {
       lung = lung1;
       lat = lat1;
   }

   double arie() { //metode
       return lung * lat;
   }

   boolean maiMare(Dreptunghi d) {
       return arie() < d.arie();
   }

   void afisare() {
       System.out.println("lungime " + lung + ",latime " + lat);
   }
}
```

Pentru a folosi clasa Dreptunghi putem adăuga în această clasă metoda main sau putem crea o nouă clasă care sa aibă doar metoda main (clasă principală). Vom alege în continuare cea de a doua variantă.

```
class DreptunghiMain {

   public static void main(String arg[]) {
       double a;
       //cream un dreptunghi
       Dreptunghi d;
       d = new Dreptunghi(3, 5);
       System.out.print("Dreptunghi initial: ");
       d.afisare();
       a = d.arie();
       System.out.println("arie=" + a);

       //cream un nou dreptunghi, de dimensiuni primite ca argumente
       int x = Integer.parseInt(arg[0]);
       int y = Integer.parseInt(arg[1]);
       Dreptunghi d2 = new Dreptunghi(x, y);
       System.out.print("Noul dreptunghi: ");
       d2.afisare();
       System.out.println("arie=" + d2.arie());
       if (d.maiMare(d2))
           System.out.println("Noul dreptunghi este mai mare");
       else
           System.out.println("Dreptunghiul initial este mai mare");
       d2 = new Dreptunghi(); //se va apela constructorul fara argumente
       System.out.print("Dreptunghi unitate: ");
       d2.afisare();
   }
}
```

Clasa Object

În Java, **orice clasă** este extinsă, nu neapărat direct, din clasa `Object`[1]. Această clasă pune la dispoziţie anumite metode, dintre amintim:

Metoda equals

- Verifică dacă un obiect este egal (în relaţie de echivalenţă) cu cel dat ca referinţă. Foarte important de reţinut este faptul că operatorul de egalitate "==" verifică **egalitatea pe referinţe**, deci `ob1 == ob2` dacă şi numai dacă cele două obiecte reprezintă aceeaşi referinţă. În practică, cel mai adesea se doreşte verificarea egalităţii câmpurilor obiectelor. În aceast caz, suprascrierea metodei equals este necesară.

Urmăriți exemplul de mai jos pentru a înțelege diferența dintre "==" și equals.

```
public class A {

   int x;
   int y;

   public A(int a, int b) {
      x = a;
      y = b;
   }

   @Override
   public boolean equals(Object obj) {
       if(obj instanceof A) {
           return this.x == ((A)obj).x && this.y == ((A)obj).y;
       }
       return false;
   }

   public static void main(String[] args) {

       A ob1 = new A(2, 3);
       A ob2 = ob1; // Aceeași referință

       System.out.println(ob1 == ob2); // va printa true

       A ob3 = new A(2, 3); // Nu e aceeași referință
       System.out.println(ob1 == ob3); // va printa false

       System.out.println(ob1.equals(ob2)); // va printa true
       System.out.println(ob1.equals(ob3)); // va printa true
   }
}
```

Supracrieți metoda equals pentru clasa Dreptunghi de mai sus și testați.

Metoda hashCode

- În Java, metoda **hashCode** este folosită pentru a codifica datele stocate într-o instanță a clasei într-o valoare hash (32-bit signed integer). De fiecare dată când se apelează pentru același obiect metoda hashCode, aceasta trebuie să întoarcă aceeași valoare.
- Dacă două obiecte sunt egale conform metodei equals, atunci metoda hashCode trebuie sa producă același rezultat pentru fiecare dintre cele două obiecte. Reciproca nu este neapărat adevărată.

Un exemplu de suprascriere a metodei hashCode pentru clasa `A` de mai sus este:

```
@Override
public int hashCode() {
    int hash = 1;
    hash = hash * 31 + x;
    hash = hash * 17 + y;
    return hash;
}
```

Metoda `toString`

- Această metodă întoarce o reprezentare a obiectului respectiv ca şir de caractere. Este foarte utilă pentru a printa stările obiectelor.

Un exemplu de suprascriere a metodei toString pentru clasa `A` de mai sus este:

```
    @Override
public String toString() {
    return "(" + x + ", " + y + ")";
}
```

Supracrieţi metoda toString pentru clasa Dreptunghi de mai sus.

Cuvântul cheie this

- Cuvântul cheie `this` se referă la instanţa curentă a clasei şi poate fi folosit de metodele (care nu sunt statice!) ale unei clase pentru a referi obiectul curent. Poate fi folosit de exemplu pentru a face diferenţa dintre câmpuri ale obiectului curent şi argumentele care au acelaşi nume.

Exemplu:

```
public class Student {

    private String name;
    private double averageGrade;

    public Student(String name, double averageGrade) {
        this.name = name;
        this.averageGrade = averageGrade;
    }
}
```

 Să se scrie o clasă `ComplexNumber` pentru lucru cu numere complexe. Clasa va avea două câmpuri, reprezentând partea reală şi partea imaginară a numărului complex şi trei constructori:

- un constructor fără argumente, care lasă partea reală şi cea imaginară zero
- un constructor cu un argument, reprezentând partea reală a numărului complex, partea imaginară fiind inițializată cu zero (pentru crearea de numere reale)
- un constructor cu două argumente, reprezentând partea reală şi partea imaginară a numărului complex.

De asemenea, clasa va avea următoarele metode:

- o metodă care returnează modulul numărului complex curent
- o metodă care returnează conjugatul numărului complex curent
- o metodă de afişare a numărului complex (vezi metoda `toString()` amintită mai sus)
- o metodă care adună la numărul complex curent un număr complex primit ca parametru (plus metode similare pentru scădere, înmulţire, împărţire)

Să se scrie apoi o clasă principală care, în metoda `main` creează două obiecte de tip Complex, le afişează, afişează modulul şi conjugatul acestora, precum şi numerele obținute adunând primul număr complex la cel de al doilea, apoi scăzând, înmulţind şi împărţind cele două numere. Creați şi afişați apoi trei numere complexe, apelând fiecare din cei trei constructori ai clasei.

*Să se creeze apoi un tablou de numere complexe de dimensiune dată de la tastatură, să se citească de la tastatură partea reală, respectiv cea imaginară a lor şi să se afişeze suma numerelor citite.

Cuvântul cheie static

- Atunci când se creează mai multe instanţe ale aceleiaşi clase, fiecare dintre acestea au propriile copii ale variabilelor membre. Modificarea stării unei instanţe nu influenţează starea altei instanţe, fiecare entitate fiind independentă una de cealaltă.

- Există, totuşi, posibilitatea ca uneori, anumite câmpuri din cadrul unei clase să aibă valori independente de instanţele acelei clase, fiind comune tuturor instanţelor. Aceste câmpuri se declară folosind cuvântul cheie `static` şi le vom denumi câmpuri statice sau variabile de clasă. Ele sunt asociate cu clasa respectivă şi nu cu un obiect anume.

- Pentru a accesa un câmp static al unei clase, presupunând că acesta nu are modificatorul `private`, se face referire la clasa din care provine, nu la vreo instanţă.


Acelaşi mecanism este disponibil şi în cazul metodelor statice.


Exemplu

```
class StaticFieldExample {

   static String staticField = "Static field in class";

   private static String privateStaticField = "Private static field in class";

   public static String getPrivateStaticField() {
       return privateStaticField;
   }
}

class StaticFieldTest {

   public static void main(String[] args) {

       System.out.println(StaticFieldExample.staticField);

       /* Câmpul static privat privateStaticField nu poate
           fi accesat direct, ci numai prin getter. */
       System.out.println(StaticFieldExample.getPrivateStaticField());
   }
}
```


Pentru clasa `ComplexNumber` scrisă anterior, adăugaţi metode statice pentru fiecare operaţie binară, care să primească două numere complexe ca argumente şi să întoarcă rezultatul sumei, al diferenţei, al înmulţirii, al împărţirii acestora.


Scrieţi o clasă `Student`(nume, prenume, medie, id). Fiecare student va avea asociat un `id`, dat de numărul său de ordine (1 pentru primul student, 2 pentru cel de-al doilea ş.a.m.d). Scrieţi un constructor clasa Student şi o metodă care afişează numărul total de studenţi.

Cuvântul cheie final - Obiecte immutable

- Variabilele declarate cu atributul `final` pot fi iniţializate o singură dată, fie printr-un constructor, fie printr-o asignare. Dacă **toate** atributele unui obiect admit o unică iniţializare, spunem că acel obiect este **immutable**, în sensul că starea lui internă nu se poate modifica.


Exemple de astfel de obiecte sunt instanţe ale claselor String şi Integer. O dată create, prelucrările asupra lor se fac prin **instanţierea de noi obiecte**, şi nu prin alterarea obiectelor în sine.

```
String s = "abc";
```

```
s.toUpperCase(); /* s-ul nu se modifica. Metoda intoarce o
      referinta catre un nou obiect */
s = s.toUpperCase(); // s este acum o referinta catre un nou obiect
```

- Unei variabile de tip referinţă care are atributul final îi poate fi asignată o singură valoare (variabila poate puncta către un singur obiect). O încercare nouă de asignare a unei astfel de variabile va avea ca efect generarea unei erori la compilare. Totuşi, obiectul către care punctează o astfel de variabilă poate fi modificat (prin apeluri de metodă sau acces la câmpuri).

Exemplu: Presupunem că avem clasele de mai jos. Ce va afişa în urma executării codului?

```
public class Position {

   double x;
   double y;

   public Position(double x, double y) {
       this.x = x;
       this.y = y;
   }

   @Override
   public String toString() {
       return "(" + x + ", " + y + ")";
   }
}

public class Circle {

   public static final double PI = 3.14; // O valoare constantă

   public final double radius; // O variabilă final

   public final Position position; // O referinţă final

   public Circle(double radius, Position position) {
       /* Se iniţializează câmpurile final.
          Orice altă iniţializare ulterioară va da
          eroare la compilare */
       this.radius = radius;
       this.position = position;
   }

   @Override
   public String toString() {
       return "Circle{" +
               "radius=" + radius +
               ", position=" + position +
```

```
                '}';
    }

    public static void main(String[] args) {

        Circle circle = new Circle(2, new Position(1, 2));
        System.out.println(circle);

        //circle.radius = 3; // Eroare!
        //circle.position = new Position(2, 3); // Eroare

        circle.position.x = 2; /* Nu dă eroare, se modifică
               obiectul către care punctează referinţa, nu
               referinţa în sine! */
        circle.position.y = 7;

        System.out.println(circle);
    }
}
```

3. Tablouri multidimensionale

În Java, **tablourile multidimensionale** trebuie gândite ca tablouri unidimensionale ale căror elemente sunt tablouri unidimensionale.

 **Exemple:**

```
int[][] a = new int[3][3];
for(int i = 0; i < 3; ++i) {
   for(int j = 0; j < 3; ++j) {
       a[i][j] = i + j;
   }
}

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

 Să se creeze o clasă `MatriceTriunghiulara` care are două câmpuri: un număr întreg n şi un tablou bidimensional a de numere întregi, precum şi două metode:

- o metodă pentru citirea matricei inferior triunghiulară a de dimensiune n
- o metodă de afişare a acestei matrice.

Să se scrie o clasă principală în care să se creeze şi să se afişeze o matrice inferior triunghiulară, folosind clasa `MatriceTriunghiulara`.

4. Exerciţii

 Să se scrie o clasă care implementează o listă simplu înlănţuită de numere întregi. Aceasta va folosi clasa `Node` (ce implementează un singur nod) şi va avea metodele:

```
addLast(int value);
addFirst(int value);
addToPosition(int value, int position);
remove(int value);
removeAll(int value);
getLength();
```

 Să se construiască o clasă Muchie ce are următoarele câmpuri: vârfIniţial, vârfFinal, cost şi doi constructori:

- unul cu două argumente pentru muchii fără ponderi; costul va fi considerat 0.
- unul cu trei argumente pentru muchii cu ponderi.

Să se citească datele dintr-un *fişier* cu structura:

```
5
2 3 10.5
1 3 4
1 2 17.23
1 4 20
3 4 0.5
```

şi să se construiască şi să se afişeze vectorul corespunzător de muchii.

 Pentru citirea din fişier, se foloseşte, de asemenea, clasa Scanner:

```
Scanner sc = new Scanner(new File("input.txt"));
```

Atenţie! Trebuie tratată excepţia `FileNotFoundException`, altfel va da eroare la compilare.

```
try {
   Scanner sc = new Scanner(new File("input.txt"));
} catch (FileNotFoundException fne) {
   System.out.println("File not found!");
}
```

 Folosind clasa scrisă anterior, scrieţi a aplicaţie pentru parcurgerea în lăţime a unui graf neorientat. Acesta va fi dat prin:

1. liste de adiacenţă
2. matrice de adiacenţă. Datele se vor citi din fişier.