Lab. 4: Moștenire. Clase abstracte.

Cuprins

1. Recapitulare laborator 3

În laboratorul 3, am făcut o introducere a claselor în Java, am văzut cum se declară şi se instanţiază. De asemenea, am prezentat clasa `Object`, din care derivă, direct sau indirect, orice altă clasă. Clasa `Object` pune la dispoziţie unele metode principale, care de cele mai multe ori trebuie suprascrise pentru a verifica de exemplu egalitatea obiectelor, a obţine o reprezentare text a obiectelor etc. Amintim `equals, hashCode, toString`.

Nu în ultimul rând, am introdus cuvântul cheie `this`, am definit variabiele şi metodele statice şi am văzut cum se pot crea obiecte imutabile (cuvântul cheie `final`). Am amintit şi tablourile multidimensionale, care practic pot fi gândite ca nişte tablouri unidimensionale ale căror elemente sunt tablouri unidimensionale.

Încercaţi să răspundeţi la următoarele întrebări:

- Cum se poate defini o clasă în Java?
- Cum se realizează instanţierea unei clase?
- Cum se poate verifica egalitatea obiectelor? Care este semnificaţia operatorului "==" ?
- La ce se referă cuvântul cheie `this`?
- Ce puteţi spune despre variabilele statice?
- Ce reprezintă un obiect immutable ?

2. Modalităţi de reutilizare a codului în Java

În Java, reutilizarea codului se face la nivel de clasă în următoarele 2 moduri:

- Agregare (compunere)
- Mostenire

Agregare (Compunere)

Agregarea reprezintă pur şi simplu prezenţa unei referinţe la un obiect într-o altă clasă. Acea clasă practic va refolosi codul din clasa corespunzătoare obiectului.

Exemplu:

```
class Page {
    private String content;
    public int no;
```

```
    public Page(String c, int no) {
        this.content = c;
        this.no = no;
    }
}

class Book {
    private Page[] pages;

    public Book(int dim, String title) {
        pages = new Page[dim];
        for (int i=0; i < dim; i++)
            pages[i] = new Page("Pagina " + i, i);
    }
}
```

Din punct de vedere conceptual, există două tipuri de agregare:

- **strong** - la dispariţia obiectelor conţinute prin agregare, existenţa obiectului container încetează (de exemplu, o carte nu poate exista fără pagini)
- **weak** - obiectul-container poate exista şi în absenţa obiectelor agregate (de exemplu, o bibliotecă poate exista şi fără cărţi).

Iniţializarea obiectelor conţinute poate fi făcută în 3 momente de timp distincte:

- la **definirea** obiectului (înaintea constructorului, fie folosind o valoare iniţială, fie blocuri de iniţializare[1])
- în cadrul **constructorului**
- chiar **înainte de folosire** (se numeşte *lazy initialization*)

Moştenire (Inheritance)

Moştenirea este un mecanism de refolosire a codului specific limbajelor orientate obiect şi reprezintă posibilitatea de a defini o clasă care **extinde** o altă clasă deja existentă. În Java, se poate extinde din cel mult **o singură clasă**. Ideea de bază este de a **prelua** funcţionalitatea existentă într-o clasă şi de a **adăuga** una nouă sau de a o **modela** pe cea existentă.

Clasa existentă este numită **clasă-părinte** (clasă de bază sau super-clasă). Clasa care extinde clasa - părinte se numeşte **clasă-copil** (child), clasă derivată sau sub-clasă.

Presupunem că avem clasa *A* deja definită şi implementată. Pentru ca o clasă *B* să derive din *A*, în Java folosim cuvântul cheie **extends**.

```
class A {
    // Membrii şi metodele declarate în A.
}

class B extends A {
    // Membrii şi metodele din A sunt mostenite.
    // Membrii şi metodele declarate, specifice lui B.
}
```

Java suportă doar moştenire publică. Astfel, toţi membrii şi toate metodele superclasei sunt moştenite şi pot fi folosite de subclasă. Singura excepţie sunt membrii privaţi ai superclasei ce **nu** pot fi accesaţi direct din subclasă. De asemenea, constructorii nu sunt consideraţi membrii ai clasei şi nu sunt moşteniti de subclasă, dar constructorul superclasei poate fi invocat din subclasă. Pentru a apela constructorul superclasei, Java pune la dispozitie cuvântul cheie **`super`**. Apelul **`super`**, dacă acesta există, trebuie să fie prima instrucţiune din constructorul subclasei.

```
class A {
    public A() {
        System.out.println("New A");
    }
}

class B extends A {
    public B() {
        super();
        System.out.println("New B");
    }
}
```

Observaţie:

- La crearea unui obiect de tipul subclasei se apelează constructori din ambele clase: mai întâi cel din superclasă şi apoi cel din subclasă.
- Dacă apelul **`super`** nu există, compilatorul va încerca să îl ataşeze pe cel fără argumente, ceea ce va cauza o eroare dacă în clasa de bază nu este definit un constructor fără argumente.

- Fiecare constructor din subclasă trebuie să aibă un constructor cu aceeaşi signatură în clasa părinte sau să apeleze explicit un alt constructor al clasei extinse.

 **Exemplu:**

```
public class ClassOne {
   int x;

   public ClassOne(int x) {
       System.out.println("ClassOne constructor");
       this.x = x;
   }
}

public class ClassTwo extends ClassOne {
   int y;

   public ClassTwo(int x, int y) {
       super(x);
       System.out.println("ClassTwo constructor");
       this.y = y;
   }
}
```

Declararea unui câmp în subclasă cu aceeaşi denumire ca a unui câmp din superclasă are ca efect **mascarea** celui din superclasă. Totuşi, acesta poate fi accesat prin intermediul **super** sau, indirect, prin getters şi setters.

Redeclararea unei metode din superclasă în subclasă se face folosind acelaşi nume ca şi în superclasă (se numeşte **override**). Aceasta are scopul de a particulariza implementările metodei respective în concordanţă cu necesităţile fiecărei clase. În plus, uneori este necesară reimplementarea pentru a folosit câmpuri declarate în clasa curentă. Prin redeclarare, codul aferent metodei din clasa părinte este ignorat dacă nu este prezent apelul **super.**

 **Observaţie:** În general putem folosi cuvântul cheie super sub una dintre formele:

- `super(...)`: pentru a invoca un constructor al superclasei clasei curente;

- `super.met(...)`: pentru a invoca o metodă a superclasei, metodă ce a fost redefinită în clasa curentă;

- `super.c`: pentru a accesa un câmp al superclasei, câmp ce a fost ascuns în clasa curentă prin redeclararea sa.

Observăm că la accesarea unui câmp sau a unei metode, `super` acționează ca referință la obiectul curent ca instanțiere a superclasei sale.

Dacă o metodă este declarată **final** în clasa părinte, aceasta **nu** poate fi redeclarată.

Agregare vs. moştenire

Când se foloseşte moştenirea şi când compunerea?

Răspunsul la această întrebare depinde, în principal, de datele problemei analizate, dar şi de concepţia designerului, neexistând o reţetă general valabilă în acest sens. În general, **compunerea** este folosită atunci când se doreşte folosirea **trăsăturilor** unei clase în interiorul altei clase, dar **nu** şi interfaţa sa (prin moştenire, noua clasă ar expune şi metodele clasei de bază). Putem distinge două cazuri largi:

- uneori se doreşte implementarea funcţionalităţii obiectului conţinut în noua clasă şi **limitarea** acţiunilor utilizatorului la metodele din noua clasă (mai exact, se doreşte să nu se permită utilizatorului folosirea metodelor din vechea clasă). Pentru a obţine acest efect se va **agrega** în noua clasă un obiect de tipul clasei conţinute şi având specificatorul de acces private.
- obiectul conţinut (agregat) trebuie/se doreşte a fi accesat **direct**. În acest caz vom folosi specificatorul de acces public. Un exemplu în acest sens ar fi o clasă numită *Maşină* care conţine ca membrii publici obiecte de tip *Motor*, *Roată* etc.

Moştenirea este un mecanism care permite crearea unor versiuni "specializate" ale unor clase existente (de bază). Aceasta este folosită în general atunci când se doreşte construirea unui tip de date care să reprezinte o implementare specifică (o specializare oferită prin clasa derivată) a unui lucru mai general. Un exemplu simplu ar fi clasa *Dacia* care moşteneşte clasa *Maşină*.

Diferenţa dintre moştenire şi agregare este de fapt diferenţa dintre cele 2 tipuri de relaţii majore prezente între obiectele unei aplicaţii:

- **is a** - indică faptul că o clasă este derivată dintr-o clasă de bază (intuitiv, dacă avem o clasă *Animal* şi o clasă *Câine*, atunci ar fi normal să avem *Câine* derivat din *Animal*, cu alte cuvinte *Câine* ***is a(n)*** *Animal*);
- **has a** - indică faptul că o clasă-container are o clasă conţinută în ea (intuitiv, dacă avem o clasă *Maşină* şi o clasă *Motor*, atunci ar fi normal să avem *Motor* referit în cadrul *Maşină*, cu alte cuvinte *Maşină* ***has a*** *Motor*).

Upcasting şi downcasting

Convertirea unei referinţe la o clasă derivată într-una a unei clase de bază ale acesteia poartă numele de **upcasting**. Upcasting-ul este făcut **automat** şi **nu** trebuie declarat explicit de către programator.

```
class Instrument {
    public void play() {}
    static void tune(Instrument i) {
        i.play();
    }
}

// Obiectele de suflat sunt instrumente si au aceeasi interfata
public class Wind extends Instrument {
    public static void main(String[] args) {
        Wind flute = new Wind();
        Instrument.tune(flute);   // Upcasting automat !!!
    }
}
```

 **Observaţie:**

- Deşi obiectul `Flute` este o instanţă a clasei `Wind`, acesta este pasat ca parametru în locul unui obiect de tip `Instrument`, care este o superclasa a clasei `Wind`.
- Upcasting-ul se face la pasarea parametrului. Termenul de **upcasting** provine din diagramele de clase (UML în special) în care moştenirea se reprezintă prin 2 blocuri aşezate unul sub altul, reprezentând cele 2 clase (sus este clasa de bază, iar jos clasa derivată), unite printr-o săgeată orientată spre clasa de bază.

Downcasting este operaţia **inversă** upcast-ului şi este o conversie explicită de tip în care se merge în **jos** pe ierarhia claselor (se converteşte o clasă de bază într-una derivată). Acest cast trebuie făcut **explicit** de către programator. Downcasting-ul este **posibil** numai dacă obiectul declarat ca fiind de o clasă de bază este, de fapt, instanţa clasei derivate către care se face downcasting-ul.

 **Exemplu:**

```
class Animal {
    public void eat() {
        System.out.println("Animal eating");
    }
}

class Wolf extends Animal {
    public void howl() {
```

```
        System.out.println("Wolf howling");
    }

    public void eat() {
        System.out.println("Wolf eating");
    }
}

class Snake extends Animal {
    public void bite() {
        System.out.println("Snake biting");
    }
}

class Test {
    public static void main(String[] args) {
        Animal a [] = new Animal[2];

        a[0] = new Wolf();
        a[1] = new Snake();

        for (int i = 0; i < a.length; i++) {
            a[i].eat();                          // 1
            if (a[i] instanceof Wolf)
                ((Wolf)a[i]).howl();             // 2
            if (a[i] instanceof Snake)
                ((Snake)a[i]).bite();            // 3
        }
    }
}
```

Observaţie:

- În liniile marcate cu **2** şi **3** se execută un downcast de la *Animal* la *Wolf*, respectiv *Snake* pentru a putea fi apelate metodele specifice definite în aceste clase.
- Înaintea execuţiei **downcast-ului** (conversia de tip la *Wolf,* respectiv *Snake*), verificăm dacă obiectul respectiv este de tipul dorit (utilizand operatorul **instanceof**). Dacă am încerca să facem downcast către tipul *Wolf* al unui obiect instanţiat la *Snake*, maşina

virtuală ar semnala acest lucru aruncând o excepţie la rularea programului (ClassCastException).

- Apelarea metodei *eat()* (linia 1) se face direct, fără downcast, deoarece această metodă este definită şi în clasa de bază. Datorită faptului că *Wolf* suprascrie (*overrides*) metoda *eat()*, apelul *a[0].eat()* va afisa *"Wolf eating"*. Apelul *a[1].eat()* va apela metoda din clasa de bază (la ieşire va fi afisat *"Animal eating"*) deoarece *a[1]* este instanţiat la *Snake*, iar *Snake* nu suprascrie metoda *eat()*.

Upcasting-ul este un element foarte important. De multe ori răspunsul la întrebarea: *este nevoie de moştenire?* este dat de răspunsul la întrebarea: *am nevoie de upcasting?* Aceasta deoarece upcasting-ul se face atunci când pentru unul sau mai multe obiecte din clase derivate se execută aceeaşi metodă definită în clasa părinte.

Clase abstracte

O clasă abstractă defineşte un model (un concept abstract) ce va fi folosit pentru definirea altor clase ce pot fi instanţiate. O clasă abstractă nu poate fi instanţiată, doar oferă un comportament comun subclaselor sale. Se defineşte folosind cuvântul cheie **`abstract`**.

Metodele care nu sunt însoţite de o implementare se numesc abstracte şi pot fi incluse doar în clasele abstracte. O metodă abstractă trebuie să fie marcată cu **`abstract`**, altfel se va genera o eroare la compilare. Fiecare subclasă a unei clase abstracte va implementa toate metodele abstracte ale acesteia.

Nu este obligatoriu ca o clasă abstractă să conţină metode abstracte, dar o clasă ce conţine cel putin o metodă abstractă este obligatoriu să fie declarată abstractă.

Exemplu:

```
abstract class GraphicObject {
    int x, y;
    void moveTo(int newX, int newY) {
        // implementare
    }
    abstract void draw();
    abstract void resize();
}
```

Implicatii ale mostenirii

- Specificatorul de access **protected**:
  - folosit în declararea unui câmp sau a unei metode dintr-o clasă - specifică faptul că membrul sau metoda respectivă poate fi accesată doar din cadrul **clasei** însăşi sau din clasele **derivate** din aceasta.
- Specificatorul de access **private**:
  - folosit în declararea unui câmp sau a unei metode dintr-o clasă - specifică faptul că membrul sau metoda respectivă poate fi accesată doar din cadrul **clasei** însăşi, **nu** şi din clasele derivate din aceasta.
- Cuvantul cheie **final**:
  - folosit la declararea unei metode, implică faptul că metoda nu poate fi suprascrisă în clasele derivate;
  - folosit la declararea unei clase, implică faptul că acea clasă nu poate fi derivată.

Observaţie: Pe lângă reutilizarea codului, moştenirea dă posibilitatea de a dezvolta pas cu pas o aplicaţie (procedeul poartă numele de *incremental development*). Astfel, putem folosi un cod deja funcţional şi adăuga alt cod nou la acesta, astfel izolându-se bug-urile în codul nou adăugat.

Exercitii

1. Întrucât în ierarhia de clase Java, clasa `Object` se află în rădăcina arborelui de moştenire pentru orice clasă, orice clasă va avea acces la o serie de facilităţi oferite de `Object`. Amintim metoda `toString()`, al cărei scop este de a oferi o reprezentare a unei instanţe de clasă sub forma unui şir de caractere.
   - Definiţi clasa **Persoana** cu un membru nume de tip `String`, şi o metodă `toString()` care va returna acest nume.
   - Clasa va avea, de asemenea:
     - un constructor fără parametri
     - un constructor ce va iniţializa numele.
   - Din ea derivaţi clasele **Profesor** şi **Student**:
     - Clasa `Profesor` va avea membrul materie de tip `String`.
     - Clasa `Student` va avea membrul nota de tip `int`.
     - Clasele vor avea:
       - constructori fără parametri

        - constructori care permit inițializarea membrilor. Identificați o modalitate de **reutilizare** a codului existent.
    - Instanțiați clasele `Profesor` și `Student`, și apelați metoda `toString()` pentru fiecare instanță.
2. Adăugati metode `toString()` în cele două clase derivate, care să returneze tipul obiectului, numele și membrul specific. De exemplu:
    - pentru clasa `Profesor`, se va afișa: "Profesor: Ionescu, POO"
    - pentru clasa `Student`, se va afișa: "Student: Popescu, 10"
3. Modificați implementarea `toString()` din clasele derivate astfel încât aceasta să utilizeze implementarea metodei `toString()` din clasa de bază.
4. Adăugați o metodă `equals` în clasa `Student`. Justificați criteriul de echivalență ales.
5. Upcasting.
    - Creați un vector de obiecte `Persoană` și populați-l cu obiecte de tip `Profesor` și `Student` (upcasting).
    - Parcurgeți acest vector și apelati metoda `toString()` pentru elementele sale.
6. Downcasting.
    - Adăugați clasei `Profesor` metoda *predă* și clasei `Student` metoda *învață*. Implementarea metodelor constă în afișarea numelui și a acțiunii.
    - Parcurgeți vectorul de la exercițiul anterior și, folosind downcasting la clasa corespunzătoare, apelați metodele specifice fiecarei clase (*predă* pentru `Profesor` și *învață* pentru `Student`). Pentru a stabili tipul obiectului curent folosiți operatorul **instanceof**.
    - Modificați programul anterior astfel încât downcast-ul să se facă mereu la clasa `Profesor`.