PROGRAMARE ORIENTATĂ PE OBIECTE

LABORATOR 11

Anul universitar 2023 – 2024

Semestrul II

1 Noțiuni teoretice

1.1 Constructori

Există uneori restricții de integritate care trebuie îndeplinite pentru crearea unui obiect. Java permite acest lucru prin existența noțiunii de constructor, împrumutată din C++. Astfel, la crearea unui obiect al unei clase se apelează automat o funcție numită constructor. Constructorul are numele clasei, nu returnează explicit un tip anume (nici măcar void) și poate avea oricâți parametri.

Crearea unui obiect se face cu sintaxa:

```
class MyClass {
   ...
}

...

MyClass instanceObject;

// constructor call
instanceObject = new MyClass(param_1, param_2, ..., param_n);
```

Se poate combina declararea unui obiect cu crearea lui propriu-zisă printr-o sintaxă de tipul:

```
// constructor call
MyClass instanceObject = new MyClass(param_1, param_2, ..., param_n);
```

De reținut că, în terminologia POO, obiectul creat în urma apelului unui constructor al unei clase poartă numele de instanță a clasei respective. Astfel, spunem că instanceObject reprezintă o instanţă a clasei MyClass.

Să urmărim în continuare codul:

```
String myFirstString, mySecondString;
myFirstString = new String();
mySecondString = "This is my second string";
```

Acesta creează întâi un obiect de tip String folosind constructorul fără parametru (alocă spațiu de memorie și efectuează inițializările specificate în codul constructorului), iar apoi creează un alt obiect de tip String pe baza unui șir de caractere constant.

Clasele pe care le-am creat până acum însă nu au avut nici un constructor. În acest caz, Java crează automat un constructor implicit (în terminologia POO, default constructor) care face inițializarea câmpurilor neinițializate, astfel:

- referințele la obiecte se inițializează cu null
- variabilele de tip numeric se inițializează cu 0
- variabilele de tip logic (boolean) se inițializează cu false

Pentru a exemplifica acest mecanism, să urmărim exemplul:

```
public class SomeClass {

    private String name = "Some Class";

    public String getName() {
        return name;
    }
}

class Test {

    public static void main(String[] args) {
        SomeClass instance = new SomeClass();
        System.out.println(instance.getName());
    }
}
```

La momentul execuției, în consolă se va afișa “Some Class” și nu se va genera nici o eroare la compilare, deși în clasa SomeClass nu am declarat explicit un constructor de forma:

```
public SomeClass() {
    ... // variables initialization
}
```

Să vedem acum un exemplu general:

```
public class Student {

    private String name;
    public int averageGrade;

    // (1) constructor without parameters
    public Student() {
        name = "Unknown";
        averageGrade = 5;
    }

    // (2) constructor with two parameters; used to set the name and the grade
    public Student(String n, int avg) {
        name = n;
```

```
            averageGrade = avg;
        }

        // (3) constructor with one parameter; used to set only the name
        public Student(String n) {
            this(n, 5); // call the second constructor (2)
        }

        // (4) setter for the field 'name'
        public void setName(String n) {
            name = n;
        }

        // (5) getter for the field 'name'
        public String getName() {
            return name;
        }
    }
```

Declararea unui obiect de tip Student se face astfel:

```
Student st;
```

Crearea unui obiect Student se face obligatoriu prin apel la unul din cei 3 constructori de mai sus:

```
st = new Student();           // first constructor call (1)
st = new Student("Gigel", 6); // second constructor call (2)
st = new Student("Gigel");    // third constructor call (3)
```

Atenție! Dacă într-o clasă se definesc doar constructori cu parametri, constructorul default, fără parametri, nu va mai fi vizibil! Exemplul următor va genera eroare la compilare:

```
class Student {

    private String name;
    public int averageGrade;

    public Student(String n, int avg) {
        name = n;
        averageGrade = avg;
    }

    public static void main(String[] args) {
        // ERROR: the implicit constructor is hidden by the constructor with parameters
        Student s = new Student();
    }
}
```

În Java, există conceptul de copy constructor, acesta reprezentând un constructor care ia ca parametru un obiect de același tip cu clasa în care se află constructorul respectiv. Cu ajutorul acestui constructor, putem să copiem obiecte, prin copierea membru cu membru în constructor.

```
public class Student {
    private String name;
    private int averageGrade;

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

    // copy constructor
    public Student(Student student) {
        // name este camp privat, noi il putem accesa direct (student.name)
        // deoarece ne aflam in interiorul clasei
        this.name = student.name;
        this.averageGrade = student.averageGrade;
    }
}
```

1.2 Referințe. Implicații în transferul de parametri

Obiectele se alocă pe heap. Pentru ca un obiect să poată fi folosit, este necesară cunoașterea adresei lui. Această adresă, așa cum știm din limbajul C, se reține într-un pointer.

Limbajul Java nu permite lucrul direct cu pointeri, deoarece s-a considerat că această facilitate introduce o complexitate prea mare, de care programatorul poate fi scutit. Totuși, în Java există noțiunea de referinţe care înlocuiesc pointerii, oferind un mecanism de gestiune transparent.

Astfel, declararea unui obiect:

```
Student st;
```

creează o referință care poate indica doar către o zonă de memorie inițializată cu patternul clasei Student fără ca memoria respectivă să conțină date utile. Astfel, dacă după declarație facem un acces la un câmp sau apelăm o funcție-membru, compilatorul va semnala o eroare, deoarece referința nu indică încă spre vreun obiect din memorie. Alocarea efectivă a memoriei și inițializarea acesteia se realizează prin apelul constructorului împreună cu cuvântul-cheie new.

Managementul transparent al pointerilor implică un proces automat de alocare și eliberare a memoriei. Eliberarea automată poartă și numele de Garbage Collection, iar pentru Java există o componentă separată a JRE-ului care se ocupă cu eliberarea memoriei ce nu mai este utilizată.

Un fapt ce merită discutat este semnificația atribuirii de referințe. În exemplul de mai jos:

```
Student s1 = new Student("Bob", 6);

Student s2 = s1;
s2.averageGrade = 10;
```

```
System.out.println(s1.averageGrade);
```

se va afișa 10.

Motivul este că s1 și s2 sunt două referințe către ACELASI obiect din memorie. Orice modificare făcută asupra acestuia prin una din referințele sale va fi vizibilă în urma accesului prin orice altă referință către el.

În concluzie, atribuirea de referințe nu creează o copie a obiectului, cum s-ar fi putut crede inițial. Efectul este asemănător cu cel al atribuirii de pointeri în C.

Transferul parametrilor la apelul de funcții este foarte important de înțeles. Astfel:

- pentru tipurile primitive se transfera prin COPIERE pe stivă: orice modificare în funcția apelată NU VA FI VIZIBILĂ în urma apelului.
- pentru tipurile definite de utilizator și instanțe de clase în general, se COPIAZĂ REFERINȚA pe stivă: referința indică spre zona de memorie a obiectului, astfel că schimbările asupra câmpurilor VOR FI VIZIBILE după apel, dar reinstanțieri (expresii de tipul: st = new Student()) în apelul funcției și modificările făcute după ele, NU VOR FI VIZIBILE după apel, deoarece ele modifică o copie a referinței originale.

```
class TestParams {

    static void changeReference(Student st) {
        st = new Student("Bob", 10);
    }

    static void changeObject(Student st) {
        st.averageGrade = 10;
    }

    public static void main(String[] args) {
        Student s = new Student("Alice", 5);
        changeReference(s);                 // apel (1)
        System.out.println(s.getName());    // "Alice"

        changeObject(s);                    // apel (2)
        System.out.println(s.averageGrade); // "10"
    }
}
```

1.3 Agregare și Compunere

Agregarea și compunerea se referă la prezența unei referințe pentru un obiect într-o altă clasă. Acea clasă practic va refolosi codul din clasa corespunzătoare obiectului.

- Agregarea (aggregation) - obiectul-container poate exista și în absența obiectelor agregate, de aceea este considerată o asociere slabă (weak association). În exemplul de mai jos, un raft de bibliotecă poate exista și fără cărți.
- Compunerea (composition) - este o agregare puternică (strong), indicând că existența unui obiect este dependentă de un alt obiect. La dispariția obiectelor conținute prin compunere, existența obiectului container încetează. În exemplul de mai jos, o carte nu poate exista fără pagini.

Inițializarea obiectelor conținute poate fi făcută în 3 momente de timp distincte:

- la definirea obiectului (înaintea constructorului: folosind fie o valoare inițială, fie blocuri de inițializare)
- în cadrul constructorului
- chiar înainte de folosire (acest mecanism se numește inițializare leneșă (lazy initialization)

Exemple de cod:

Compunere:

```
public class Foo {
    // Obiectul de tip Bar nu poate exista dacă obiectul Foo nu există
    private Bar bar = new Bar();
}
```

Agregare:

```
public class Foo {
    private Bar bar;

    // Obiectul de tip Bar poate continua să existe chiar dacă obiectul Foo nu există
    Foo(Bar bar) {
       this.bar = bar;
    }
}
```

Exemplu practic:

```
class Page {
    private String content;
    public int numberOfPages;

    public Page(String content, int numberOfPages) {
       this.content        = content;
       this.numberOfPages  = numberOfPages;
    }
}
```

```
class Book {
    private String title;                  // Compunere
    private Page[] pages;                  // Compunere
    private LibraryRow libraryRow = null;  // Agregare

    public Book(int size, String title, LibraryRow libraryRow) {
        this.libraryRow = libraryRow;
        this.title = title;

        pages = new Page[size];

        for (int i = 0; i < size; i++) {
           pages[i] = new Page("Page " + i, i);
        }
    }
}

class LibraryRow {
    private String rowName = null;         // Agregare

    public LibraryRow(String rowName) {
        this.rowName = rowName;
    }
}

class Library {

    public static void main(String[] args) {
        LibraryRow row = new LibraryRow("a1");
        Book book = new Book(100, "title", row);

        // După ce nu mai există nici o referință la obiectul Carte,
        // Garbage Collector-ul va șterge (la un moment dat, nu
        // neapărat imediat) acea instanță, dar obiectul LibraryRow
        // transmis constructorului nu este afectat.

        book = null;
    }
}
```

1.4 Moștenire (Inheritance)

Numită și derivare, moștenirea este un mecanism de refolosire a codului specific limbajelor orientate obiect și reprezintă posibilitatea de a defini o clasă care extinde o altă clasă deja existentă. Ideea de bază este de a prelua funcționalitatea existentă într-o clasă și de a adăuga una nouă sau de a o modela pe cea existentă.

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

Spre deosebire de C++, Java nu permite moștenire multiplă (multiple inheritance), astfel că nu putem întâlni ambiguități de genul Problema Rombului / Diamond Problem. Mereu când vom vrea să ne referim la metoda părinte (folosind cuvântul cheie super, cum vom vedea mai jos), acel părinte este unic determinat.

Când se folosește moștenirea și când agregarea?

Răspunsul la această întrebare depinde, în principal, de datele problemei analizate dar și de concepția designerului, neexistând o rețetă general valabilă în acest sens. În general, agregarea este folosită atunci când se dorește folosirea trăsăturilor unei clase în interiorul altei clase, dar nu și interfața sa (prin moștenire, noua clasă ar expune și metodele clasei de bază). Putem distinge două cazuri:

- uneori se dorește implementarea funcționalității obiectului conținut în noua clasă și limitarea acțiunilor utilizatorului doar la metodele din noua clasă (mai exact, se dorește să nu se permită utilizatorului folosirea metodelor din vechea clasă). Pentru a obține acest efect se va agrega în noua clasă un obiect de tipul clasei conținute și având specificatorul de acces private.
- obiectul conținut (agregat) trebuie/se dorește a fi accesat direct. În acest caz vom folosi specificatorul de acces public. Un exemplu în acest sens ar fi o clasă numită Car care conține ca membrii publici obiecte de tip Engine, Wheel etc.

Moștenirea este un mecanism care permite crearea unor versiuni "specializate" ale unor clase existente (de bază). Moștenirea este folosită în general atunci când se dorește 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 moștenește clasa Car.

Diferența dintre moștenire și agregare este de fapt diferența dintre cele 2 tipuri de relații majore prezente între obiectele unei aplicații :

- is a - indică faptul că o clasă este derivată dintr-o clasă de bază (intuitiv, dacă avem o clasă Animal și o clasă Dog, atunci ar fi normal să avem Dog derivat din Animal, cu alte cuvinte Dog is an Animal)
- has a - indică faptul că o clasă-container are o clasă conținută în ea (intuitiv, dacă avem o clasă Car și o clasă Engine, atunci ar fi normal să avem Engine referit în cadrul Car, cu alte cuvinte Car has a Engine)

1.5 Upcasting și Downcasting

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

Exemplu de upcasting:

```
class Instrument {
    public void play() {}

    static void tune(Instrument i) {
        i.play();
    }
}
// Obiectele Wind sunt instrumente
// deoarece au ca și clasa-parinte clasa Instrument
public class Wind extends Instrument {
    public static void main(String[] args) {
        Wind flute = new Wind();
        Instrument.tune(flute); // !! Upcasting automat pentru că metoda primește
                                // un obiect de tip Instrument, nu un obiect de tip Wind
                                // Deci ar fi redundant să faci un cast explicit cum ar fi:
                                // Instrument.tune((Instrument) flute)
    }
}
```

Deși obiectul flute este o instanță 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 (în special UML) în care moștenirea se reprezintă prin 2 blocuri așezate 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 operația inversă upcast-ului și este o conversie explicită de tip în care se merge în jos pe ierarhia claselor (se convertește 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, instanță clasei derivate către care se face downcasting-ul.

Iată un exemplu în care este folosit downcasting:

```
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");
    }
}

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

        animals[0] = new Wolf();    // Upcasting automat
        animals[1] = new Snake();   // Upcasting automat

        for (int i = 0; i < animals.length; i++) {
            animals[i].eat(); // 1

            if (animals[i] instanceof Wolf) {
                ((Wolf)animals[i]).howl(); // 2
            }

            if (animals[i] instanceof Snake) {
                ((Snake)animals[i]).bite(); // 3
            }
        }
    }
}
```

Codul va afișa:

```
Wolf eating
Wolf howling
Animal eating
Snake biting
```

În liniile marcate cu 2 și 3 se execută un downcast de la Animal la Wolf, respectiv Snake pentru a putea fi apelate metodele specifice definite în aceste clase. Înaintea execuției downcast-ului (conversia de tip la Wolf respectiv Snake) verificăm dacă obiectul respectiv este de tipul dorit (utilizând operatorul instanceof). Dacă am încerca să facem downcast către tipul Wolf al unui obiect instanțiat la Snake mașina virtuală ar semnala acest lucru aruncând o excepție la rularea programului.

Apelarea metodei eat() (linia 1) se face direct, fără downcast, deoarece această metodă este definită și în clasa de bază Animal. Datorită faptului că Wolf suprascrie (overrides) metoda eat(), apelul a[0].eat() va afișa “Wolf eating”. Apelul a[1].eat() va apela metoda din clasă de bază (la ieșire va fi afișat “Animal eating”) deoarece a[1] este instanțiat la Snake, iar Snake nu suprascrie metoda eat().

2 Aplicații

2.1 Produs

Implementați o aplicație de gestionare a produselor care să conțină două clase principale: Product și ProductManager. Clasa Product trebuie să aibă câmpurile name (nume) și price (preț), precum și metodele necesare pentru inițializarea, accesarea și modificarea acestor câmpuri. Clasa ProductManager trebuie să conțină metoda main în care sunt create și gestionate exemple ale clasei Product. Această metodă trebuie să demonstreze crearea, afișarea și modificarea prețului unor produse fictive.

2.2 Muncitori

Implementați un program Java pentru gestionarea salariilor angajaților. Aveți trei clase: Muncitor, Sef, și CEO. Clasa Muncitor are un singur câmp h (ore lucrate) și o metodă salariu() care calculează salariul în funcție de numărul de ore lucrate (salariul orar este de 2). Clasa Sef este derivată din Muncitor și overridează metoda salariu() pentru a calcula salariul șefului (salariul orar este de 5). Clasa CEO este derivată din Sef și overridează din nou metoda salariu() pentru a calcula salariul CEO-ului (salariul orar este de 10).

Programul trebuie să ceară utilizatorului să introducă tipul de muncitor (1 pentru muncitor, 2 pentru șef, 3 pentru CEO) și numărul de ore lucrate. Apoi, programul ar trebui să creeze un obiect corespunzător tipului de muncitor și să afișeze salariul acestuia.

3 Rezolvări

3.1 Produs

```
package com.mycompany.productmanager;

class Product {
    private String name;
    private double price;

    public Product(String name, double price) {
        this.name = name;
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    @Override
    public String toString() {
        return "Product{name='" + name + "', price=" + price + '}';
    }
}
public class ProductManager {
    public static void main(String[] args) {
        // Exemplu de utilizare a clasei Product
        Product product1 = new Product("Laptop", 1500.0);
        Product product2 = new Product("Mouse", 25.0);

        // Afișare informații despre produse
        System.out.println("Product 1: " + product1);
        System.out.println("Product 2: " + product2);

        // Modificare prețului produsului 1 și afișare
        product1.setPrice(1600.0);
        System.out.println("Updated price of Product 1: " + product1.getPrice());
    }
}
```

3.2 Muncitori

```
package com.mycompany.muncitori;

import java.util.Scanner;
```

```
class Muncitor
{
    protected int h;  // ore lucrate

    Muncitor(int h)
    {
        this.h = h;
    }

    double salariu()
    {
        return h * 2;
    }
}

class Sef extends Muncitor
{
    Sef(int h)
    {
        super(h);
    }

    double salariu()
    {
        return h * 5;
    }
}

class CEO extends Sef
{
    CEO(int h)
    {
        super(h);
    }

    double salariu()
    {
        return h * 10;
    }
}


public class Muncitori {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        Muncitor m = null;

        System.out.println("Introduceti tipul muncitorului:");
        int tip = scanner.nextInt();
        System.out.println("Introduceti numarul de ore muncite:");
        int ore = scanner.nextInt();
```

```
        System.out.println(tip);

        switch(tip){
            case 1: m = new Muncitor(ore); break;
            case 2: m = new Sef(ore); break;
            case 3: m = new CEO(ore); break;
            default: m = new Muncitor(ore);
        }

        System.out.println(m.salariu());
    }
}
```

4 Teme

Denumiți proiectele realizate de voi cu denumirea problemei și numele vostru. Exemplu: ***matrice_Bold Nicolae***.

4.1 Complex

Implementați o aplicație Java pentru gestionarea numerelor complexe (transpuneți problema Complex de la C++ pentru limbajul Java). Clasa Complex are două câmpuri, r (partea reală) și i (partea imaginară). Ea conține constructori pentru inițializarea numerelor complexe, o metodă afis() pentru afișarea unui număr complex, o metodă read () pentru citirea de la tastatură a unui număr complex, precum și metode pentru adunarea a doi numere complexe, incrementarea cu 1 a unui număr complex și verificarea dacă un număr complex este mai mare decât altul.

Programul trebuie să ceară utilizatorului să introducă două numere complexe și să afișeze suma lor, rezultatul incrementării fiecărui număr complex cu 1, suma unui număr complex cu o valoare constantă, și să determine care dintre cele două numere complexe este mai mare."