Laborator 3: Clase și obiecte

1 Responsabil

Gabriel Guțu-Robu, gabriel.gutu@upb.ro
Publicat: 22 octombrie 2022, 18:15

2 Clase și obiecte

2.1 Definiție clasă și obiect

Un *obiect* reprezintă o entitate din lumea reală care poate fi identificat în mod distinct (ex.: student, mașină, carte, formă geometrică, etc.) Orice obiect se caracterizează prin: **identitate, stare** și **comportament**.

Starea unui obiect cuprinde *proprietăți* sau *câmpuri*. Un pătrat are un câmp numit latură, ce reprezintă o proprietate caracteristică obiectului.

Comportamentul unui obiect este cuprins în *acțiunile* sale, definite de metode. Invocarea unei metode presupune a cere obiectului să efectueze o anumită acțiune (ex.: metoda *aria*() care calculează aria unui pătrat).

Obiectele de același tip sunt definite utilizându-se o clasă comună. O clasă este un *contract* care definește câmpurile și metodele obiectelor. Se pot crea mai multe instanțe ale unei clase, procedeu numit *instanțiere*.

O clasă Java utilizează *variabile* pentru a defini câmpurile și *metode* pentru a defini acțiunile. Crearea de noi obiecte se realizează cu ajutorul unor metode speciale numite **constructori**, ce inițializează câmpurile obiectelor.

Clasa `Patrat` reprezintă o definiție pentru obiecte de tipul pătrat, având un constructor cu parametri și unul fără parametri.

```
class Patrat {
      double latura = 2.0;
      Patrat() {}
      Patrat(double laturaNoua) {
            this.latura = laturaNoua;
      }
      double aria() {
            return this.latura * this.latura;
      }
}
```

Această clasă nu conține metoda `main`. Ea poate fi definită fie în aceeași clasă (caz în care clasa `Patrat` trebuie să fie publică), fie separat în altă clasă, numită clasa `TestPatrat`:

```
public class TestPatrat {
	public static void main(String[] args){
		Patrat p1 = new Patrat();	//latura = 2.0
		System.out.println("Aria1 = " + p1.aria());
		Patrat p2 = new Patrat(5.0);	//latura = 5.0
		System.out.println("Aria2 = " + p2.aria());
	}
}
```

În urma rulării clasei `TestPatrat` se va afișa:

```
Aria1 = 4.0
Aria2 = 25.0
```

Se pot pune ambele clase într-un fișier, însă numai o clasă poate fi publică. **Clasa publică trebuie să aibă același nume cu numele fișierului**. Astfel, fișierul se va numi `TestPatrat.java`.

Operatorul `new` este folosit pentru a crea noi obiecte cu ajutorul constructorului. Sunt create două obiecte de tipul pătrat – unul cu latura de valoare implicită `2.0` și unul cu latura de `5.0`, valoare primită ca parametru în constructor. Prin apelul metodei `aria()`, atașat numelui obiectului, se calculează aria. Putem accesa și câmpul `latura` cu ajutorul referințelor: `p1.latura` și `p2.latura`.

Metoda `main` poate fi încorporată în cadrul clasei `Patrat`, la începutul sau la sfârșitul acesteia. Așadar, existența clasei `TestPatrat` care conține metoda `main` nu este neapărat necesară.

```
class Patrat {
	double latura = 2.0;
	Patrat() {
	}
	Patrat(double laturaNoua) {
		this.latura = laturaNoua;
	}
	double aria() {
		return this.latura * this.latura;
	}
	public static void main (String[] args){
		Patrat p1 = new Patrat();
		System.out.println("Aria1 = " + p1.aria());
		Patrat p2 = new Patrat(5.0);
		System.out.println("Aria2 = " + p2.aria());
	}
}
```

Un constructor are următoarele proprietăți:

- Are același nume cu clasa;
- Nu returnează nimic (nici măcar tipul `void`);
- Este invocat cu ajutorul operatorului `new` și are rol în inițializarea obiectelor;

- Poate fi **supraîncărcat** – adică metode cu același nume, dar parametri de tipuri diferite;
- public `void Patrat { }` este o metodă oarecare ce returnează tipul `void` și nu constructorul clasei.

O clasă poate fi definită fără constructor. În acest caz, un *constructor implicit* (fără argumente) este furnizat automat clasei.

Obiectele sunt accesate utilizând *variabile de tipul referință*. O clasă este un tip referință, ceea ce înseamnă că o variabilă de tipul clasei poate referenția o instanță a clasei.

```
NumeClasa referintaObiect;
```

Următoarea instrucțiune declară variabila `p` de tipul `Patrat`, creează un obiect cu ajutorul operatorului `new` și asignează referința lui `p`:

```
Patrat p = new Patrat ();
```

Obiectul și variabila de tip referință sunt două noțiuni distincte, dar care uneori se întrepătrund în practică. Este mai ușor să zicem că `p` este un `Patrat` decât că `p` este o variabilă ce conține o referință către un obiect de tipul `Patrat`.

Tablourile sunt văzute ca obiecte în Java și se instanțiază cu ajutorul operatorului `new`. O variabilă tablou conține o referință la un tablou.

Accesarea câmpurilor și metodelor se face cu ajutorului operatorului de acces (.). Pentru exemplul de mai sus, câmpul `latura` poate fi accesat prin `p.latura`, iar metoda `aria()`, prin `p.aria()`. Se poate crea un obiect și fără a-l asigna explicit unei variabile. În exemplul de mai jos se creează un obiect anonim:

```
System.out.println("Aria = " + new Patrat(5).aria());
```

Atunci când o referință a unui obiect este neinițializată, ea va stoca valoarea `null`. Câmpurile (sau variabilele membru) ale unei clase neinițializate iau valorile: `0` pentru tipul numeric, `false` pentru `boolean` și `'\u000'` pentru tipul `char`. Java nu asignează nicio valoare implicită variabilelor locale dintr-o metodă. În cazul de mai jos, Java va semnala eroare de compilare pentru variabila neinițializată `x`.

```
public static void main (String[] args){
      int x;
      System.out.println(x);
}
```

Excepția `NullPointerException` apare atunci când se apelează o metodă pentru o variabilă referință ce stochează valoarea `null`. Înainte de a apela o metodă, asigurați-vă că variabila stochează referința la un obiect.

Pentru o variabilă de tip primitiv, valoarea acelei variabile este de tip primitiv. Pentru o variabilă de tip referință, valoarea este o referință către locația obiectului.

Fie următorul exemplu:

```
Patrat p1 = new Patrat();
Patrat p2 = new Patrat(3.0);
p1 = p2;
```

A treia instrucțiune copiază referința `p2` în `p1`. După atribuire, `p1` și `p2` vor referi același obiect. Obiectul referențiat înainte de `p1` nu mai este folositor, fiind cunoscut ca *garbage*. *Java runtime system* identifică situațiile de garbage și eliberează spațiul ocupat de aceste variabile. Procesul este cunoscut sub numele de *garbage collection*. Dacă nu mai este nevoie de un anumit obiect, se poate asigna `null` referinței acelui obiect. JVM va colecta automat spațiul ocupat de un obiect care nu este referențiat de către nicio variabilă. În exemplul de mai sus, `p1` și `p2` au locații de memorie diferite până când este suprascrisă locația lui `p1`.

În situația în care se dorește ca toate instanțele unei clase să partajeze anumite informații, se folosesc *variabilele statice*, numite și *variabile de clasă*. Variabilele statice partajează aceeași zonă de memorie, în așa fel încât dacă un obiect le schimbă tipul sau valoarea, acest lucru va fi vizibil pentru toate celelalte obiecte ale clasei. În Java se pot declara și defini atât variabile, cât și metode statice. Ele pot fi accesate fără a folosi o instanță a clasei.

Constantele unei clase sunt partajate de toate obiectele clasei. Pentru aceasta, ele trebuie declarate `final static`.

```
public class Cerc {
	double raza;
	static int nrObiecte = 0;
	final static double PI = 3.1415;
	Cerc(double razaNoua){
		this.raza = razaNoua;
		nrObiecte++;
	}
	public static int getNrObiecte() {
		return nrObiecte;
	}
	double aria(){
		return raza * raza * PI;
	}
	public static void main(String[] args){
		System.out.println("Numar obiecte = " + Cerc.getNrObiecte());
		Cerc c1 = new Cerc(2.0);
		Cerc c2 = new Cerc(3.0);
		System.out.println("Numar obiecte = " + Cerc.getNrObiecte());
	}
}
```

Atât variabila `nrObiecte`, cât și metoda `getNrObiecte()` sunt statice, astfel că pot fi apelate cu ajutorul numelui clasei, sub forma `Cerc.getNrObiecte()` și `Cerc.nrObiecte`.

Variabilele și metodele statice pot fi folosite în metode statice sau în metode de instanță. Simetric nu este valabil – deci variabilele și metodele de instanță, care aparțin unui anumit

obiect, pot fi apelate doar în metode de instanță, nu și în metode statice. În schimb, acestea pot fi accesate prin intermediul unui obiect din clasă. Fie următorul exemplu:

```
public class Dreptunghi {
	double latime;
	double lungime;

	public static void main(String[] args) {
		Dreptunghi d = new Dreptunghi();
		double l = d.latime;
		double L = lungime;
	}
}
```

Instrucțiunea `double l = d.latime;` este corectă, pe când instrucțiunea `double L = lungime;` nu este corectă. Prin urmare, alegeți să definiți ca statice variabilele și metodele care nu depind de o instanță a clasei, ci mai degrabă sunt valabile întregii clase.

O variabilă membru a clasei poate avea același nume cu o variabilă locală a unei metode, fără să se creeze confuzie (a se vedea următorul exemplu).

```
public class Clasa1 {
	private int a = 2;
	private int b = 2;

	public void numeMetoda() {
		int a = 5;
		System.out.println(a);  //va afișa 5
		System.out.println(b);  //va afișa 2
	}
}
```

Apelul metodei `numeMetoda()` va afișa valorile `5` și `2`. Ultima valoare asignată variabilei `a` este referențiată în instrucțiunea `System.out.println(a)`.

Pornind de la următorul exemplu:

```
public class Patrat {
	double latura;

	public Patrat (double latura) {
		this.latura = latura;
	}

	public Patrat() {
		this(5.0);
	}
}
```

Instrucțiunea `this.latura` se referă la câmpul `latura` al obiectului curent. Apelul `this(5.0)` se adresează constructorului cu parametri `public Patrat (double latura)` și are ca scop modificarea atributului acestuia.

2.2 Tipuri primitive și obiecte

Tipurile de date primitive nu sunt tratate ca obiecte în Java. Multe metode Java însă necesită utilizarea obiectelor ca argumente. De exemplu, metoda `add` din clasa `ArrayList` adaugă un obiect într-un `ArrayList`. Java oferă o modalitate convenabilă de a încorpora tipuri primitive în obiecte și de a folosi astfel avantajele programării generice. Spre exemplu, clasa wrapper pentru tipul `int` este `Integer`, iar pentru tipul `double`, clasa `Double`.

Clasele `Boolean, Character, Double, Float, Byte, Short, Integer` și `Long` sunt wrappere pentru tipurile de date primitive corespunzătoare. Ele sunt grupate în pachetul `java.lang. Double, Float, Long, Integer, Short` și `Byte`, extind clasa abstractă `Number` și au metodele `doubleValue(), floatValue(), longValue(), intValue(), shortValue(), byteValue()`, ce convertesc obiectele la tipuri primitive specifice. `Character` și `Boolean` sunt moștenite direct din clasa `Object`. Toate suprascriu metodele `toString()` și `equals()` din clasa `Object` și implementează interfața `Comparable`, de unde este suprascrisă metoda `compareTo()`.

Toate clasele wrapper sunt **immutable**, deci valoarea obiectelor nu poate fi modificată. Ele nu au constructori fără parametri. Pentru constructorii cu parametri, argumentul poate fi atât o valoare numerică, cât și un string – `new Integer(10)`, dar și `new Integer("10")`.

Pentru a returna valoarea unui obiect Integer:

```
Integer myInt = new Integer(15);
int i = myInt.intValue();
System.out.println("Valoarea lui i = " + i);
```

Pentru a inițializa un obiect cu valoarea reprezentată de un string:

```
Integer ob = Integer.valueOf("15");
```

`Integer.parseInt()` convertește un `String` la tipul `int`.

`Integer.parseInt("10")` va returna `10`

`Integer.parseInt("11", 2)` va returna `3` - este reprezentarea lui `11` în baza de numerație `2`.

Metodele de conversie pentru `Integer` și `Float` sunt prezentate mai jos. Similar există pentru `Byte, Short, Long` și `Double`.

```
public static int parseInt(String s)
public static int parseInt(String s, int radix)
public static float parseFloat(String s)
public static float parseFloat(String s, int radix)
```

Un tablou de obiecte care provin dintr-o clasă ce implementează interfața `Comparable` poate fi sortat cu ajutorul metodei `sort()` din pachetul `java.util.Arrays`. Întrucât interfața `Comparable` este implementată, programatorul poate defini o metodă de sortare la alegere, dacă nu dorește să folosească `java.util.Arrays.sort`. Un tabou este o instanță a clasei `Object`.

```
Double[] tablou = {new Double(5.5), new Double(1.5), new Double(8.9) };
java.util.Arrays.sort(tablou);
```

Definiția `Double obj = new Double(5.5);` este echivalentă cu `Double obj = 5.5;`

Un tablou de obiecte din clasa `Double` poate fi inițializat astfel:

```
Double[] tablou = {5.5, 9.8, -6.7};
```

Pentru a lucra cu numere întregi sau reale foarte mari, se pot folosi clasele `BigInteger` și `BigDecimal` din pachetul `java.math`. Ambele sunt immutable, extind clasa `Number` și implementează interfața `Comparable`.

2.3 Modificatorii de acces

Clasele și funcțiile menționate până acum au fost declarate utilizând un specificator special: `public`. În limbajul Java (și în majoritatea limbajelor de programare de tipul POO), orice clasă, atribut sau metodă posedă un specificator de acces, al cărui rol este de a restricționa accesul la entitatea respectivă. Există specificatorii:

- `public` - permite acces complet din exteriorul clasei curente;
- `private` - limitează accesul doar în cadrul clasei curente.

Folosit în declararea unui atribut sau a unei metode dintr-o clasă, specifică faptul că atributul sau metoda respectivă poate fi accesată doar din cadrul clasei, nu și din clasele derivate din aceasta.

- `protected` - limitează accesul doar în cadrul clasei curente și al tuturor descendenților ei.

Folosit în declararea unui atribut sau a unei metode dintr-o clasă, specifică faptul că atributul sau metoda respectivă poate fi accesată doar din cadrul clasei sau din clasele derivate din aceasta.

- `default` - în cazul în care nu este utilizat explicit nici unul din specificatorii de acces de mai sus, accesul este permis doar în cadrul pachetului (`package private`).

Atenție, nu confundati specificatorul `default` cu `protected`.

- cuvântul cheie `final`.

Pentru a preveni modificarea atributelor unei clasei din afara ei, se utilizează modificatorul `private`. Acest procedeu poartă denumirea de **încapsularea datelor**. Pentru a accesa câmpul privat se folosește o metodă publică `get` care returnează valoarea atributului, iar pentru a-l modifica, o metodă `set`.

Dacă este folosit la declararea unei metode, indică faptul că metoda nu poate fi suprascrisă în clasele derivate. Dacă este folosit la declararea unei clase, indică faptul că acea clasă nu poate fi derivată.

Utilizarea specificatorilor contribuie la realizarea încapsulării. Amintim că, încapsularea se referă la acumularea atributelor și metodelor caracteristice unei anumite categorii de obiecte într-o clasă. Pe de altă parte, acest concept denotă și ascunderea informației de stare a unui obiect, reprezentată de atributele acestuia, alături de valorile aferente, și asigurarea comunicării strict prin intermediul metodelor (interfața clasei). Acest lucru conduce la izolarea modului de implementare al unei clase (atributele acesteia și felul în care metodele le manipulează) de utilizarea acesteia. Utilizatorii unei clase pot conta pe funcționalitatea expusă de aceasta, indiferent de modalitatea în care ea este implementată, aceasta putându-se chiar modifica în timp. Accesul utilizatorilor la implementarea unei clase ar conduce la imposibilitatea modificării acesteia, fără a declanșa actualizări unde clasa este folosită.

3 Aplicații

3.1 Clasa de studenți

a. Creați o clasă **Student** care va conține următoarele proprietăți:

- nrMatricol (de tip `long`)
- nume (de tip `String`)
- prenume (de tip `String`)
- medieFinala (de tip `double`)

Implementați metode de tipul *set* și *get* pentru variabilele membru.

b. Implementați o metodă în clasa **Student** denumită `detaliiStudent()` care va întoarce o valoare de tip `String` cu detaliile studentului de forma:

```
[NrMatricol] Prenume Nume: mediaFinala
```

c. În metoda **main** creați 3 instanțe ale clasei `Student` și populați-le cu date folosind metodele `set` create anterior.

```
Student student1 = new Student();
student1.setNrMatricol(1);
student1.setNume("Ion");
student1.setPrenume("Ionescu");
student1.setMedieFinala(9.8);
```

d. Creați un vector (tablou) denumit `semigrupa`, care va stoca instanțele de tipul `Student` definite anterior. Afișați detaliile despre studenți folosind metoda

`detaliiStudent`().

e. Modificați media unuia dintre studenții din vector alegând un index aleator (folosind metoda **random** din clasa **Math**) și afișați din nou vectorul `semigrupa`.
f. Calculați media semigrupei si afișați rezultatul.

3.2 La doctor

- Creați clasele:
  - `Doctor` cu variabila membru:
    - `nume` de tipul `String`.
  - `Pacient` cu variabilele membru:
    - `nume` de tipul `String`
    - `id` de tipul `Integer`

a. Variabilele vor avea modificatorul de acces `private`;
b. Cele două clase vor conține constructori fără parametri și constructori cu parametri;
c. Adăugați metode de tip `get` și `set` pentru fiecare variabilă;
d. Modificați clasa `Doctor` astfel încât numele să nu poată fi schimbat după instanțiere.
e. Modificați clasa `Pacient` astfel încât variabila `id` să fie generata automat, incremental la fiecare instanțiere a clasei.
f. Adăugați o metodă `afisare()` in clasa `Pacient`, ce va întoarce un `String` ce conține id-ul si numele pacientului. Instanțiați un vector de tip `Pacient` cu 3 instanțe.
g. Afișați la consola rezultatul metodei `afisare()` pentru fiecare pacient din vectorul declarat anterior. Verificați dacă id-ul este incrementat la fiecare instanțiere.

3.3 Medici de familie

h. Adăugați un câmp `rating` în clasa `Doctor` de tip `Integer[]` și un câmp `medieRating`, ce va stoca media rating-urilor.
i. Creați o metodă `adaugaRating`, ce va adăuga un rating doctorului și va recalcula media.
j. Modificați clasa `Doctor` astfel încât să încorporeze un câmp `pacienti` de tip `Pacient[]`. Creați metode specifice de asignare a pacienților unui doctor.
k. Adăugați o metoda `afisare()` in clasa `Doctor`, ce va afișa numele doctorului, rating-ul și datele pacienților asignați lui.