

Laborator 5: Clase abstracte și interfețe

1 Responsabil

Gabriel Guțu-Robu, gabriel.gutu@upb.ro

Publicat: 06 noiembrie 2022, 08:42

2 Clase abstracte

Clasele abstracte sunt clase din care **NU** se pot crea instanțe cu ajutorul operatorului `new`. O *metodă abstractă* este definită fără implementare (se definește doar antetul ei). Implementarea unei metode abstracte se face în clasele derivate (care o extind). O clasă care conține o metodă abstractă trebuie declarată abstractă, dar ea poate conține și metode care nu sunt abstracte (care au o implementare). Constructorul din clasa abstractă poate avea orice modificador de acces, însă este recomandat să fie *protected* pentru a putea fi utilizat în clasele copil (care o extind). Când se creează o instanță a unei subclase, constructorul superclasei este invocat (așa cum știm deja până acum).

```
public abstract class ObiectGeometric {  
    private String culoare = "alb";  
    protected ObiectGeometric() {  
    }  
    protected ObiectGeometric(String culoare) {  
        this.culoare = culoare;  
    }  
    public abstract double getAria();  
}
```

```
class Cerc extends ObiectGeometric{  
    private double raza;  
    public Cerc() {  
    }  
    public Cerc(double raza) {  
        this.raza = raza;  
    }  
    public Cerc(double raza, String culoare) {  
        super(culoare);  
        this.raza = raza;  
    }  
    public double getRaza() {  
        return raza;  
    }  
}
```

```

    public void setRaza(double raza) {
        this.raza = raza;
    }
    public double getAria() {
        return raza * raza * Math.PI;
    }
}

```

- O metodă abstractă nu poate fi conținută într-o clasă care nu este abstractă. Dacă o subclasă a unei superclase abstracte nu implementează toate metodele abstracte, atunci ea trebuie declarată abstractă. Toate metodele abstracte sunt non-stactice;
- O clasă abstractă NU poate fi instanțiată folosind operatorul `new`, însă poate avea constructori, care vor fi invocați de constructorii subclaselor;
- O clasă care conține metode abstracte trebuie să fie abstractă. Este posibil să se definească o clasă abstractă care nu conține metode abstracte;
- O subclasă poate fi abstractă, chiar dacă superclasa ei este non-abstractă;
- O subclasă poate suprascrie o metodă din superclasă și să o definească abstractă. În acest caz, subclasa trebuie definită abstractă;
- Nu se poate crea o instanță din clasa abstractă folosind operatorul `new`, dar o clasă abstractă poate fi folosită ca un tip de date (la declararea unui obiect).

```

ObiectGeometric[] obiecte = new ObiectGeometric[10];
obiecte[0] = new Cerc();

```

3 Interfețe

O interfață conține doar constante și metode abstracte. O interfață se declară astfel:

```

modifier_acces interface numeInterfata {
}

```

O interfață este tratată ca o clasă specială în Java. Fiecare interfață este compilată într-un fișier bytecode separat, la fel ca orice altă clasă. Din interfețe **NU** se pot crea instanțe folosind operatorul `new`, dar ele funcționează pe linia moștenirii, la fel ca și clasele abstracte.

Toate câmpurile unei interfețe sunt `public static final`, iar **metodele** sunt `public abstract` (nu au implementare). O constantă poate fi accesată astfel: `numeInterfata.NUME_CONSTANTA`.

Interfețele respectă următoarele reguli:

- Antet: `interface numeInterfata;`
- Pot conține doar antete de metode și constante;
- Relația respectată este IS-A;

- Nu se pot instanția sau implementa, ci doar implementa (*implements*);
- Interfața poate să extindă doar altă interfață și nimic altceva (*extends*);
- Interfața nu poate să implementeze nici un tip de obiect;
- Scopul principal al utilizării interfețelor este organizarea și definirea modelului pe care dorim să îl implementăm;
- Interfața este folosită pentru a descrie un protocol între clase: **o clasă care implementează o interfață va implementa metodele definite în interfață**. Astfel, orice cod care folosește o anumită interfață știe ce metode pot fi apelate pentru acea interfață;
- Interfața poate fi declarată public doar dacă este definită într-un fișier cu același nume ca și interfața. Dacă o interfață nu este declarată public, atunci modificatorul ei de acces este *default* (*package-private*);
- Pentru a defini o clasă ce implementează o interfață, folosim cuvântul cheie `implements`.
- După ce o interfață a fost implementată, acea implementare devine o clasă obișnuită care poate fi extinsă prin moștenire;
- O clasă (non abstractă) **poate să implementeze mai multe interfețe** (deși, ne reamintim, poate extinde o singură clasă).

4 Comparație clase abstracte și interfețe

O clasă poate moșteni o singură clasă de bază, dar poate implementa mai multe interfețe. O interfață poate moșteni alte interfețe (dar nu și clase), utilizând cuvântul cheie `extends`. O astfel de interfață se mai poate numi sub-interfață. Toate clasele au ca rădăcină și clasă de bază principală clasa `Object`, însă acest principiu nu se aplică interfețelor. Se poate spune că o relație puternică de tipul *is-a* determină folosirea moștenirii și a claselor de bază, iar o relație mai slabă de tipul *is-a* determină implementarea interfețelor.

În general, se preferă utilizarea interfețelor, deoarece acestea sunt mai flexibile și definesc un super-tip comun pentru clase între care nu există legături. Mai mult, implementarea unor interfețe permite implementarea unor comportamente diferite în cadrul unor clase.

Tabelul 1 prezintă o comparație între clase abstracte și interfețe.

Tabelul 1. Clase abstracte și interfețe.

	Variabile	Constructori	Metode
Clase abstracte	Fără restricții	Nu se pot crea obiecte folosind operatorul <i>new</i> .	Fără restricții

		Constructorii sunt apelați prin subclase.	
Interfețe	Toate variabilele trebuie să fie <i>public</i> <i>static final</i>	Nu se pot crea obiecte folosind operatorul <i>new</i> . Nu au constructori.	Metodele trebuie să fie public abstract.

5 Interfața Comparable

Interfața Comparable, care compară două obiecte, este definită astfel:

```
package java.lang;
public interface Comparable {
    public int compareTo(Object o)
}
```

Metoda `compareTo` determină ordinea obiectului curent față de obiectul `o` și returnează un întreg negativ, zero sau un întreg pozitiv dacă obiectul curent este mai mic, egal sau mai mare ca `o`.

Multe clase din Java (de exemplu `String` sau `Date`) implementează interfața `Comparable`. Putem spune că un obiect din clasa `String` de exemplu, este o instanță nu doar a lui `String`, ci și a lui `Object` și `Comparable`.

```
public class String extends Object implements Comparable {
    // corpul clasei
}
public class Date extends Object implements Comparable {
    // corpul clasei
}
```

Metoda `min` din clasa `Min` determină minimumul a două obiecte în două moduri.

```
public class Min {
    public static Comparable min (Comparable o1, Comparable
o2 ) {
        if (o1.compareTo(o2) < 0)
            return o1;
        else
            return o2;
    }
}
```

sau

```
public class Min {
    public static Object min (Object o1, Object o2 ) {
        if (((Comparable)o1).compareTo(o2) < 0)
            return o1;
        else
            return o2;
    }
}

public static void main(String[] args) {
    String s1 = "Ana";
    String s2 = "Maria";

    String s3 = (String)Min.min(s1, s2);
    System.out.println(s3);
}
```

La rularea metodei main se va afișa:

Ana

În secvența de mai jos este implementată clasa CercComparabil, ce suprascrie metoda compareTo() pentru a compara ariile a două cercuri. Se face downcasting cu conversie explicită de la tipul Object la tipul CercComparabil.

```
public class CercComparabil extends Cerc implements
Comparable {
    public CercComparabil(double raza) {
        super(raza);
    }
    public int compareTo(Object o) {
        if (getAria() > ((CercComparabil)o).getAria())
            return 1;
        else
            if (getAria() < ((CercComparabil)o).getAria())
                return -1;
            else
                return 0;
    }
}
```

O metodă din clasa Object este equals, cu următorul antet:

```
public boolean equals(Object o)
```

Metoda testează dacă două obiecte sunt identice, la apelul:

```
obiect1.equals(obiect2);
```

Implementarea metodei equals în clasa Object este:

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

Această implementare testează dacă cele două referințe pointează la același obiect, folosind operatorul `==`. În clasele derivate, metoda `equals` se poate suprascrie astfel încât să verifice dacă două obiecte au conținut identic. Prototipul metodei, `public boolean equals(Object o)`, trebuie să se păstreze și la suprascriere. Exemplu pentru clasa `Cerc`:

```
public boolean equals(Object o) {
    if (o instanceof Cerc) {
        return raza == ((Cerc)o).raza;
    }
    return false;
}
```

Metoda `compareTo` nu este definită în clasa `Object`. Ea face parte din interfața `Comparable` și permite compararea obiectelor care sunt instanțe ale claselor ce implementează `Comparable`. Este de dorit ca `equals` și `compareTo` să fie consistente, adică dacă `o1.compareTo(o2) == 0`, atunci `o1.equals(o2) == true`.

6 Interfața Cloneable

Interfața `Cloneable` permite copierea obiectelor. Definirea acestei interfețe este vidă, adică nu conține metode abstracte sau constante, motiv pentru care este numită și *marker interface*. Pentru o clasă care implementează interfața `Cloneable`, obiectele pot fi copiate utilizând metoda `clone()` din clasa `Object`. Aceasta creează un nou obiect care este o clonă a obiectului inițial.

```
public class CercClonabil extends Cerc implements Cloneable,
Comparable {
    public CercClonabil(double raza) {
        super(raza);
    }
    public int compareTo(Object o) {
        if (getAria() > ((CercClonabil)o).getAria())
            return 1;
        else if (getAria() < ((CercClonabil)o).getAria())
            return -1;
        else
            return 0;
    }
}
```

```

    public Object clone() throws CloneNotSupportedException
    {
        return super.clone();
    }
}
CercClonabil c1 = new CercClonabil(5.0);
CercClonabil c2 = (CercClonabil)c1.clone();

```

Apelul `super.clone()` se referă la invocarea metodei `clone()` din clasa `Object`. `c1` și `c2` sunt obiecte cu referințe diferite, dar conținut identic. Metoda `clone()` copiază toate câmpurile din `c1` – dacă acestea sunt de tip primitiv li se copiază valoarea (**deep copy**), iar dacă sunt obiecte, li se copiază referința (**shallow copy**) – în `c2`. Dacă `CercClonabil` nu ar suprascrie metoda `clone()`, s-ar primi o eroare de sintaxă, deoarece `clone()` este declarată `protected` în clasa `Object`. Dacă `CercClonabil` nu ar implementa interfața `Cloneable`, invocarea lui `super.clone()` ar genera `CloneNotSupportedException`.

7 Aplicații

Scrieți un program care recomandă cărți în funcție de vârsta cititorilor. Implementați următoarele clase/interfețe:

7.1 Clasa Book

Clasa `Book` va avea câmpurile private: `String title`, `String[] authors`, `int year`, `String publisher`, `int recommendedAge`, `float price`.

- Adăugați un constructor cu parametri, corespunzător câmpurilor `title`, `year`, `publisher`, `recommendedAge` și `price`. Vectorul `authors` va fi de dimensiune 3;
- Adăugați metode `getter` și `setter` pentru `title`, `year`, `publisher`, `recommendedAge` și `price`. Pentru `authors` adăugați doar `getter`;
- Creați o metodă `addAuthor` care adaugă un autor la lista de autori ai unei cărți; În cazul în care vectorul de autori este plin, se va dubla capacitatea lui. *Hint: folosiți clonarea pentru a copia datele din vechiul vector!*
- Implementați metoda `toString`.

7.2 Clasa User

Clasa abstractă `User` va avea cu câmpurile: `String name`, `int age`

- Adăugați un constructor cu parametri, corespunzător câmpurilor `name` și `age`.
- Adăugați metode `getter` și `setter` pentru `name` și `age`.

7.3 Clasele Student, Teacher și Librarian

Clasele `Student`, `Teacher` și `Librarian` moștenesc clasa `User`.

7.4 Clasa Library

Clasa `Library` va conține un vector de cărți și un vector de utilizatori.

- Implementați metoda `Book[] readBooks()` care citește cărți de la tastatură până când se introduce un `String` gol, adică se apasă tasta ENTER;
- Implementați metoda `User[] readUsers()` care citește utilizatori de la tastatură până când se introduce un `String` gol, adică se apasă tasta ENTER;
- Implementați o metodă `main` în care să testați funcționalitatea aplicației.

7.5 Interfața Buyer

Interfața `Buyer` va conține metoda `float buyBook(User user)`. Clasa `Book` va implementa interfața `Buyer`.

- `buyBook` va aplica un discount pentru prețul cărții în funcție de tipul de utilizator: 5% pentru `Student`, 10% pentru `Teacher` și 20% pentru `Librarian`.

7.6 Interfața AgeRecommendation

Interfața `AgeRecommendation` va conține metodele `Book[] sortBooks()` și `Book[] recommendBooksForUser (User user)`. Clasa `Library` va implementa această interfață.

- Metoda `sortBooks()` întoarce o listă de cărți sortate crescător după `recommendedAge`. În cazul în care două cărți au aceeași `recommendedAge`, se vor sorta în ordine crescătoare după anul de apariție și în ordine alfabetică după titlu; *Hint: folosiți `Comparable` pentru a compara cărțile după anul de apariție și după titlu!*
- Metoda `recommendBooksForUser(User user)` întoarce o listă de cărți sortate după aceleași principii, dar restrânsă în funcție de vârsta utilizatorului primit ca parametru.