Lecţia 7

Interfeţe

Să presupunem că trebuie să scriem o clasă ce foloseşte la gestionarea jucătorilor unei echipe de fotbal. O parte din această clasă este prezentată mai jos.

```
class EchipaFotbal {

    private String nume;
    private JucatorFotbal capitan;
    private JucatorFotbal[] jucatori = new JucatorFotbal[10];
    ...
}
```

Ce poate fi entitatea *JucatorFotbal*? Ea poate fi o clasă (posibil subclasă a unei alte clase şi/sau abstractă) sau o *interfaţă*. Dacă despre clase, subclase şi clase abstracte s-a vorbit pe parcursul mai multor lecţii anterioare, în această lecţie se va vorbi despre interfeţe.

7.1 Definirea unei interfeţe. Componenţa unei interfeţe

Principial, o interfaţă poate fi văzută ca o clasă pur abstractă, ca o clasă ce conţine doar metode abstracte. Definirea unei interfeţe se face asemănător cu a unei clase înlocuind cuvântul cheie *class* cu cuvântul cheie *interface*.

```
interface JucatorFotbal {
    ...
}
```

Spre deosebire de o clasă, o interfaţă poate conţine doar atribute declarate *static final* şi doar metode ale căror corpuri sunt vide. În acelaşi timp, membrii unei interfeţe sunt doar publici.

```
interface JucatorFotbal {
    int categorie = 5;
    void joacaFotbal();
}
```

În exemplul de mai sus membrii interfeţei nu au fost declaraţi explicit ca fiind publici, deoarece acest aspect este implicit atunci când definim o interfaţă. Cu alte cuvinte exemplul de mai jos este identic cu cel de sus. La fel se întâmplă şi cu câmpurile care sunt implicit declarate *final static*.

```
interface JucatorFotbal {
    public int categorie = 5;
    public void joacaFotbal();
}
```

Important Toţi membrii unei interfeţe sunt publici chiar dacă specificatorul de acces *public* este omis în definirea interfeţei. O interfaţă nu poate conţine membrii declaraţi *private* sau *protected*.

Important Toate atributele unei interfeţe sunt *static final* chiar dacă ele nu sunt declarate astfel în definirea interfeţei. Datorită acestui fapt orice atribut al unei interfeţe trebuie iniţializat.

Important O interfaţă poate extinde oricâte alte interfeţe utilizând cuvântul cheie *extends*.

7.2 Implementarea unei interfeţe

După cum spuneam anterior, o interfaţă poate fi văzută ca fiind o clasă pur abstractă. După cum ştim, o clasă abstractă nu poate fi instanţiată şi deci e normal ca acest lucru să fie valabil şi în cazul interfeţelor. În acest context, ce poate referi *jf* din declaraţia de mai jos?

```
JucatorFotbal jf;
```

Referinţa *jf* poate referi orice obiect instanţă a unei clase care *implementează* interfaţa *JucatorFotbal*. A implementa o anumită interfaţă înseamnă a realiza într-un anumit mod funcţionalitatea precizată de interfaţa respectivă.

În Java, implementarea unei interfeţe de către o clasă se face folosind cuvântul cheie *implements* urmat de numele interfeţei implementate, ca în exemplele de mai jos.

```
class JucatorBunFotbal implements JucatorFotbal {

    public void joacaFotbal() {
        System.out.println("Eu joc bine fotbal.");
    }
}
```

În UML, implementarea de către o clasă a unei interfeţe se numeşte *realizare*. În Figura 7.1 se exemplifică modul de reprezentare a interfeţelor şi a relaţiilor de realizare pe baza codului prezentat anterior.

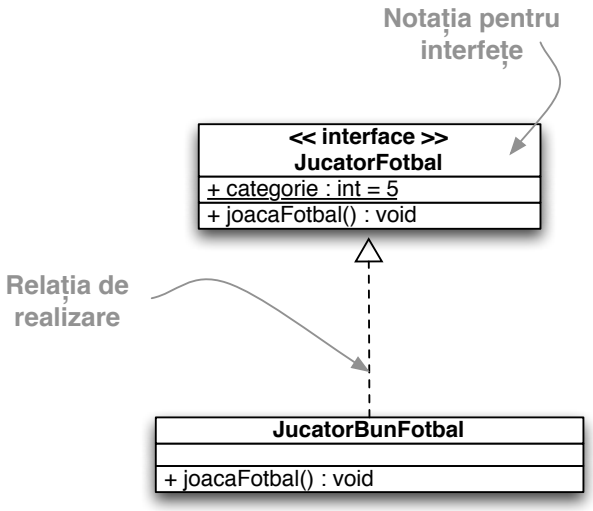


Figura 7.1: Reprezentarea UML a interfeţei şi a relaţiei de realizare.

Important În momentul în care o clasă implementează o interfaţă aceasta fie implementează toate metodele din interfaţă, fie e declarată abstractă.

Important O interfaţă poate avea oricâte implementări. Două clase distincte pot implementa o aceeaşi interfaţă.

Important O clasă poate extinde o singură clasă dar poate implementa oricâte interfeţe.

Important Dacă în clasa *JucatorFoarteBun* de mai jos specificatorul de acces al metodei nu ar fi fost public, compilatorul ar fi semnalat o eroare la compilare. Motivul este foarte simplu: în momentul în care clasa implementează o interfaţă,

ea este obligată să furnizeze clienţilor toate serviciile specificate de interfaţa implementată.

```
class JucatorFoarteBunFotbal implements JucatorFotbal {

    public void joacaFotbal() {
        System.out.println("Eu joc foarte bine fotbal.");
    }
}

interface JucatorTenis {
    void joacaTenis();
}

class JucatorFotbalTenis implements JucatorFotbal, JucatorTenis {

    public void joacaFotbal() {
        System.out.println("Eu joc fotbal.");
    }

    public void joacaTenis() {
        System.out.println("Eu joc tenis.");
    }
}

class JucatorFoarteBunFotbalTenis
    extends JucatorFoarteBunFotbal implements JucatorTenis {

    public void joacaTenis() {
        System.out.println("Eu joc tenis.");
    }
}
```

Având în vedere cele precizate mai sus, este momentul să ataşăm referinţei *jf* diferite obiecte.

```
//jf refera instanta unei clase care implementeaza interfata JucatorFotbal
jf = new JucatorBunFotbal();

//jf refera instanta unei alte clase care implementeaza JucatorFotbal
jf = new JucatorFoarteBunFotbal();

//jf refera instanta unei clase care implementeaza interfetele
//JucatorFotbal si JucatorTenis
jf = new JucatorFotbalTenis();
jf.joacaFotbal();
```

```
//Prin intermediul referintei jf pot fi accesate doar serviciile
//definite de interfata JucatorFotbal
jf.joacaTenis(); //EROARE
((JucatorFotbalTenis)jf).joacaTenis(); //CORECT dar nu se recomanda

//jf refera o instanta a unei clase ce extinde o clasa
//ce implementeaza interfata JucatorFotbal
jf = new JucatorFoarteBunFotbalTenis();
```

În toate exemplele anterioare ceea ce e important constă în faptul că *jf* poate referi o instanţă a oricărei clase ce implementează interfaţa *JucatorFotbal*, o instanţă care pune la dispoziţie serviciile din interfaţa *JucatorFotbal*.

7.3 Tipurile unui obiect

În Lecţia 1 spuneam că ansamblul tuturor operaţiilor primitive oferite de un obiect se numeşte interfaţa obiectului respectiv şi că numele unei interfeţe denotă un tip. Tot atunci am precizat că e posibil ca un obiect să aibe mai multe tipuri, acest aspect fiind delimitat de perspectiva din care este văzut obiectul. Până acum am văzut situaţii în care un obiect avea mai multe tipuri diferite: o instanţă a unei clase avea atât tipul subclasei cât şi al tuturor superclaselor sau supertipurilor acelei clase. Deoarece în Java o clasă poate extinde cel mult o clasă, utilizând doar clase, între oricare două tipuri a unui obiect există tot timpul o relaţie supertip-subtip. Datorită faptului că o clasă poate implementa mai multe interfeţe, putem modela şi obiecte ce au două sau mai multe tipuri între care nu există o astfel de relaţie.

Din punctul de vedere al organizatorilor unui campionat de fotbal este necesar ca toţi participanţii să ştie să joace fotbal, nefiind necesară practicarea de către aceştia şi a altor sporturi. În acelaşi timp, la un campionat de fotbal pot participa şi jucători de fotbal ce ştiu juca şi tenis, acest aspect rămânând necunoscut organizatorilor.

În exemplul de mai jos, referinţei *JucatorFotbal jf* i s-a ataşat un obiect instanţă a unei clase ce implementează, pe lângă interfaţa *JucatorFotbal* şi interfaţa *JucatorTenis*. Utilizând această referintă putem apela metode (servicii) definite în interfaţa *JucatorFotbal*, dar în nici un caz nu putem apela direct metoda *joacaTenis*. Acest lucru se datorează faptului că referinţa e de tip *JucatorFotbal* iar compilatorul vede că această interfaţă nu defineşte un seviciu *joacaTenis* semnalând o eroare. Dacă totuşi dorim acest lucru, va trebui să utilizăm operatorul de *cast*, convertind referinţa într-o referintă de tipul *JucatorTenis*. Acest lucru nu e indicat a se realiza deoarece dacă obiectul referit de *jf* nu ştie "juca tenis" (clasa sa nu implementează interfaţa *JucatorTenis* dar implementează *JucatorFotbal* sau altfel spus obiectul referit nu e de tip *JucatorTenis* dar e

de tip *JucatorFotbal*) se va genera o eroare la execuţia programului.

```
jf = new JucatorFotbalTenis();
jf.joacaFotbal();        //Apel corect
jf.joacaTenis();         //Eroare de compilare
((JucatorTenis)jf).joacaTenis(); //Corect, dar poate fi foarte riscant
```

Un obiect este de un anumit tip dacă acesta furnizează toate serviciile specificate de acel tip. Un obiect instanţă a clasei *JucatorFotbalTenis* are tipurile: *JucatorFotbalTenis*, *JucatorFotbal*, *JucatorTenis* şi, evident, *Object*.

7.4 Conflicte

Având în vedere că o clasă poate implementa oricâte interfeţe, e posibil să apară uneori conflicte între membrii diferitelor interfeţe implementate de o clasă. La compilare primului exemplu de mai jos va fi semnalată o eroare deoarece în cadrul metodei *oMetoda* nu se ştie care atribut C este utilizat. În cazul celui de-al doilea exemplu situaţia e asemănătoare.

```
interface A {
    int C = 5;
}

interface B {
    int C = 5;
}

class Clasa implements A,B {

    public void oMetoda(){
        int c = C + 5; //EROARE, dar se poate rezolva
                       //scriind A.C sau B.C functie de
                       //care camp C se doreste a fi utilizat
    }
}

interface AA {
    void f();
}
```

```
interface BB {
    int f();
}

class CC implements AA,BB {
    //Conflict - doua metode nu pot diferi doar prin tipul returnat
}
```

7.5 Clase abstracte versus interfeţe

Când e mai bine să folosim clase abstracte şi când interfeţe? În această secţiune vom prezenta comparativ avantajele şi dezavantajele folosirii uneia versus celeilalte.

Presupunem că vrem să descriem mai multe instrumente muzicale, printre care se află şi instrumentele vioară şi pian. Cu ajutorul oricărui tip de instrument muzical se poate cânta iar într-o orchestră pot să apară la un moment dat diferite tipuri de instrumente. În acest context pentru descrierea instrumentelor într-un limbaj de programare orientat pe obiecte putem avea o entitate numită *Instrument*. În Java, această entitate poate fi atât o clasă abstractă cât şi o interfaţă.

```
abstract class AInstrument {
    public abstract void canta();
}

interface IInstrument {
    void canta();
}
```

Definirea instrumentelor vioară şi pian, atât pentru cazul în care se extinde o clasă abstractă cât şi pentru cazul în care se implementează o interfaţă e prezentată mai jos.

```
class AVioara extends AInstrument {

    public void canta() {
        System.out.println("Vioara canta.");
    }
}

class IVioara implements IInstrument {

    public void canta() {
        System.out.println("Vioara canta.");
    }
}
```

Dar după o anumită perioadă de timp, toate instrumentele care apar pe piaţă ştiu să se acordeze automat. Este evident că toate instrumentele trebuie să furnizeze o metodă *public void acordeaza()* orchestrei din care fac parte.

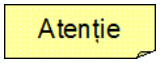
```
class APian extends AInstrument {

    public void canta() {
        System.out.println("Pianul canta.");
    }
}

class IPian implements IInstrument {

    public void canta() {
        System.out.println("Pianul canta.");
    }
}
```

Pentru varianta în care se extinde o clasă abstractă, modificările necesare sunt minime. E nevoie de adăugarea unei metode care să furnizeze serviciul pentru acordarea automată în clasa de bază, clasele *Avioara* şi *APian* rămânând nemodificate.

```
abstract class AInstrument {

    public abstract void canta();

    public void acordeaza() {
        System.out.println("Acordare automata.");
    }
}

interface IInstrument {
    void canta();
    void acordeaza();
}

class IVioara implements IInstrument {

    public void canta() {
     System.out.println("Vioara canta.");
    }

    public void acordeaza() {
     System.out.println("Acordare automata.");
    }
}
```

Pentru varianta în care se implementează o interfaţă, modificările necesare sunt mai mari decât în cazul anterior. În primul rând trebuie adăugată metoda *public void acordeaza()* în interfaţa *IInstrument* iar apoi această metodă trebuie implementată în clasele *IVioara* şi *IPian*.

```
class IPian implements IInstrument {

    public void canta() {
        System.out.println("Pianul canta.");
    }

    public void acordeaza() {
        System.out.println("Acordare automata.");
    }
}
```



Introducerea unei metode într-o interfaţă necesită modificarea tuturor claselor neabstracte care implementează respectiva interfaţă.

7.6 Exerciţiu rezolvat

Expresii Derivate

Într-un sistem de prelucrare de expresii matematice (evident simplificat) există mai multe feluri de expresii ce vor fi detaliate mai jos. Indiferent de felul concret al unui obiect expresie, pe acesta se poate apela din exteriorul său o metodă denumită calculDerivată care întoarce o referinţă la un alt obiect expresie reprezentând derivata de ordinul întâi a expresiei pe care s-a apelat metoda sus numită. Modul de construire a expresiei derivate se prezintă în detaliu mai jos împreună cu diferitele feluri de expresii:

1. Constantă - reprezintă o expresie constantă (ex. 1). Valoarea constantei este dată ca parametru la crearea unui astfel de obiect şi va fi păstrată intern de obiectul constantă creat. Derivata unei constante este o expresie constantă a cărei valoare este 0. Clasa mai defineşte o metodă ce întoarce un String care e reprezentarea şir de caractere a expresiei (ex. “1”).
2. Variabilă - reprezentând variabila “x”. Derivata unei expresii variabilă este o expresie constantă a cărei valoare este 1. Clasa mai defineşte o metodă ce întoarce un String care e reprezentarea şir de caractere a expresiei (în acest caz tot timpul “x”).
3. Sumă - reprezentând o expresie sumă între două expresii de orice fel (ex. 1 + x). Expresiile care sunt însumate vor fi date ca argumente la crearea unui obiect sumă şi vor fi memorate intern de obiectul creat. Derivata unei sume este o expresie construită după formula (a + b)’ = a’ + b’. Clasa mai defineşte o metodă ce întoarce un String care e reprezentarea şir de caractere a expresiei. Aceasta e

formată din reprezentarea String a expresiei din stânga sumei urmată de "+" şi apoi de reprezentarea String a expresiei din dreapta. Pentru a evita probleme de precedenţă a operatorilor, reprezentarea ca şir de caractere se va pune între paranteze (ex. "(1+x)").

4. Inmulţire - reprezentând o expresie înmulţire între două expresii de orice fel (ex. 2 * x). Expresiile care sunt înmulţite vor fi date ca argumente la crearea unui astfel de obiect şi vor fi memorate intern de către acesta. Derivata unei înmuliri este o expresie construită după formula (a*b)' = a'*b + a*b'. Clasa mai defineşte o metodă ce întoarce reprezentarea sub formă de String a expresiei. Aceasta e construită ca în cazul sumei dar cu semnul "*" în loc de "+" (ex. "(2 * x)")

   Notă: Nu trebuie să efectuaţi simplificări matematice în expresia derivată.

Se cere:

1. Diagrama UML detaliată a claselor prezentate, împreună cu eventuale alte clase / interfeţe care se consideră necesare.
2. Implementarea tuturor claselor.
3. Pentru exemplificare, se va construi într-o metodă main structura de obiecte corespunzătoare expresiei matematice "1 + x*x". După construcţie se va tipări pe ecran expresia, derivata de ordinul întâi al expresiei şi derivata de ordinul doi a aceleiaşi expresii.

Rezolvare

Din specificaţiile problemei (cât şi din cunoştinţele noastre de matematică) putem observa că o expresie sumă poate reprezenta suma a două alte (sub) expresii de orice fel. Astfel putem avea o sumă între două constante, între două variabile, între două expresii ce reprezintă înmulţiri, între o constantă şi o altă sumă, etc. În mod similar, putem avea diferite combinaţii de feluri de operanzi şi în cazul înmulţirii.

În plus, dacă ne gândim cum s-ar putea extinde pe viitor această aplicaţie, este clar că una din direcţiile de îmbunătăţire foarte probabile constă în adăugarea de noi feluri de expresii precum împărţire, radical, etc. La rândul lor, aceste feluri de expresii vor trebui să poată avea orice fel de operanzi şi vor trebui să poată fi incluse în sume, înmulţiri, etc. Prin urmare, numărul de combinaţii pe care va trebui să-l gestionăm pe viitor va fi cu mult mai mare decât putem observa acum într-o primă variantă a programului (număr care şi acum este destul de mare).

Pentru a putea rezolva simplu această explozie de combinaţii ne vom baza pe polimorfism, ne vom baza pe faptul că putem trata uniform diversele feluri de expresii matematice. Mai exact, dorim să putem declara variabile referinţă ce să poată referi uniform atât constante cât şi sume, înmulţiri şi variabile. Prin urmare, toate aceste obiecte vor

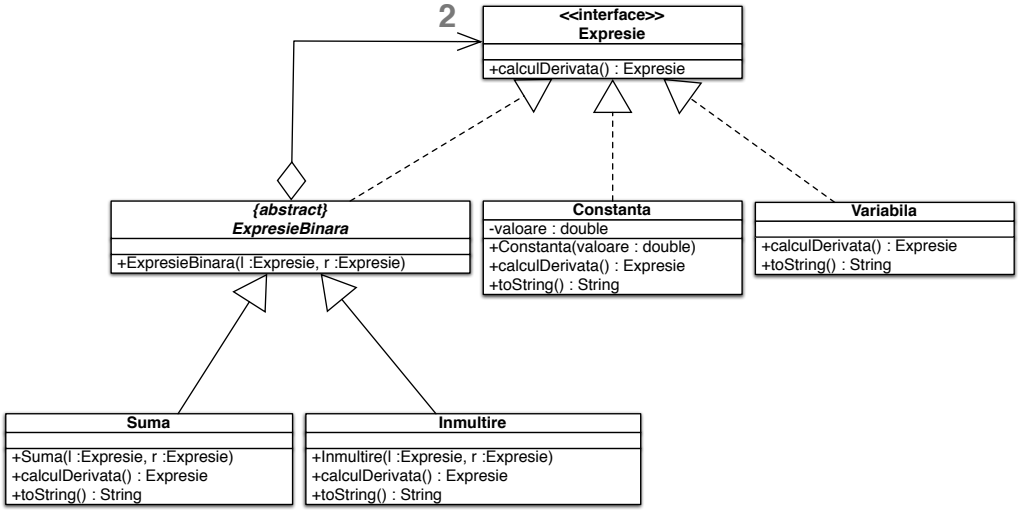


Figura 7.2: Diagrama de clase.

trebui să aibă un supertip comun pentru a putea declara referinţe cu o astfel de proprietate.

În acelaşi timp, dacă ne gândim la formula de derivare a unei sume, putem observa că, pentru a construi derivata sumei, avem nevoie de expresiile derivate ale operanzilor săi. Acelaşi lucru poate fi observat şi în cazul înmulţirii. Pe de altă parte, anterior am spus că o sumă/înmulţire poate avea ca operanzi expresii de diverse feluri (adică constante, variabile, etc.). În consecinţă, suma şi înmulţirea vor trebui să poată calcula uniform derivata operanzilor lor, vor trebui să poată apela uniform metoda *calculDerivata* indiferent de felul concret al unei expresii operand.

Având în vedere toate aceste observaţii am definit interfaţa *Expresie* ce va fi implementată de toate clasele ce modelează obiecte expresii matematice. Interfaţa conţine metoda *calculDerivata* care va întoarce o referinţă de tip *Expresie*. Motivul utilizării acestui tip returnat e simplu: derivata unei expresii oarecare poate fi o sumă de alte subexpresii, poate fi o constantă, etc. În general ar putea fi orice fel de expresie.

```
interface Expresie {

    Expresie calculDerivata();

}
```

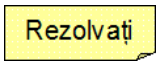
Implementarea clasei *Constanta* este extrem de simplă şi prin urmare nu o vom discuta în detaliu. Singurul lucru mai interesant constă în calculul derivatei sale. În acest sens se va crea o instanţă a clasei *Constanta* reprezentând valoarea 0, iar metoda *calculDerivata* întoarce o referinţă către acest obiect nou creat.

```
class Constanta implements Expresie {

    private double valoare;

    public Constanta(double valoare) {
        this.valoare = valoare;
    }

    public Expresie calculDerivata() {
        return new Constanta(0);
    }

    public String toString() {
        return valoare + "";
    }

}
```

În mod similar se va calcula şi derivata unei variabile: se va crea o instanţă a clasei *Constanta* reprezentând valoarea 1, iar metoda *calculDerivata* întoarce o referinţă către acest obiect nou creat.

```
class Variabila implements Expresie {

    public Expresie calculDerivata() {
        return new Constanta(1);
    }

    public String toString() {
        return "x";
    }

}
```

Din specificaţiile problemei putem observa că suma si înmulţirea au în comun un lucru: ambele expresii au doi operanzi. Prin urmare ne-am bazat pe moştenirea de clasă şi am factorizat aceste date într-o clasă abstractă *ExpresieBinara*. Ea implementează interfaţa *Expresie* şi va fi extinsă atât de clasa *Suma* cât şi de clasa *Inmultire*.



Clasa *ExpresieBinara* poate factoriza într-o formă comună şi implementarea metodei toString pe care fiecare subclasă o redefineşte. Această

factorizare nu a fost realizată aici fiind lăsată ca exerciţiu.

De ce este clasa *ExpresieBinara* declarată ca fiind abstractă? Deoarece nu are sens să o putem instanţia, rolul său fiind doar factorizarea de cod comun. Nu avem obiecte care sunt pur şi simplu expresii binare. Avem fie sume fie înmulţiri.

```
abstract class ExpresieBinara implements Expresie {

    protected Expresie st,dr;

    public ExpresieBinara(Expresie st, Expresie dr) {
        this.st = st;
        this.dr = dr;
    }
}
```

O primă sarcină care trebuie realizată de implementarea clasei *Suma* constă în setarea operanzilor conform specificaţiilor problemei. Acest lucru se realizează prin constructorul clasei care, la rândul său, memorează operanzii obiectului după cum impune superclasa (adică prin apelarea constructorului din superclasa *ExpresieBinara*).

Un alt lucru pe care îl vom discuta se referă la modul în care se va construi expresia derivată a unei sume. Astfel, metoda *calculDerivata* va crea un nou obiect sumă care are ca şi operanzi derivata operandului din stânga sumei, respectiv derivata operandului din dreapta. Derivatele operanzilor se determină simplu: prin apelarea pe obiectele corespunzătoare a metodei *calculDerivata*. În cele din urmă, metoda *calculDerivata* din clasa *Suma* întoarce o referintă la obiectul sumă nou creat. Se poate observa că modul de construcţie al derivatei reflectă exact formula de derivare cunoscută.

```
class Suma extends ExpresieBinara {

    public Suma(Expresie st, Expresie dr) {
        super(st,dr);
    }

    public Expresie calculDerivata() {
        return new Suma(st.calculDerivata(),dr.calculDerivata());
    }

    public String toString() {
        return "(" + st.toString() + " + " + dr.toString() + ")";
    }

}
```

Modul de implementare a clasei *Inmultire* este similar cu a clasei *Suma*. Din acest motiv nu vom mai descrie implementarea obiectelor înmulţire.

```
class Inmultire extends ExpresieBinara {

    public Inmultire(Expresie st, Expresie dr) {
        super(st,dr);
    }

    public Expresie calculDerivata() {
        Expresie t1 = new Inmultire(st,dr.calculDerivata());
        Expresie t2 = new Inmultire(st.calculDerivata(),dr);
        return new Suma(t1,t2);
    }

    public String toString() {
        return "(" + st.toString() + " * " + dr.toString() + ")";
    }

}
```

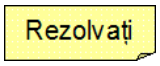
În metoda main construim în primul rând expresia cerută. Astfel, vom crea un obiect constantă reprezentând valoarea 1 (referită de *c1*), două variabile (referite de *v1* respectiv *v2*), o înmultire (referită de *i1*) care va avea ca operanzi variabilele create anterior (referite de *v1* respectiv *v2*) şi, în final, o sumă (referită de *exp*) ce are ca operanzi constanta (*c1*) respectiv suma creată anterior (referită de *i1*).

În continuare tipărim pe ecran expresia. Se poate observa că pur şi simplu tipărim referinţa *exp*. Acest lucru e posibil deoarece pentru toate clasele am redefinit corespunzător metoda *toString* declarată în clasa *Object*.

Pentru calculul derivatei de ordinul unu vom apela pe referinţa *exp* metoda *calculDerivata* şi salvăm valoarea întoarsă în variabila *deriv1* (care este tot o expresie). Pentru calculul derivatei de ordinul doi se va apela *calculDerivata* pe *deriv1* (adică pe derivata de ordiul unu). Evident, se tipăresc aceste derivate conform cerinţelor.

```
class Main {

    public static void main(String[] args) {
        Constanta c1 = new Constanta(1);
        Variabila v1 = new Variabila();
        Variabila v2 = new Variabila();
        Inmultire i1 = new Inmultire(v1,v2);
        Suma exp = new Suma(c1,i1);
```

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

        Expresie deriv1 = exp.calculDerivata();
        System.out.println(deriv1);

        Expresie deriv2 = deriv1.calculDerivata();
        System.out.println(deriv2);
    }
}
```



Rezolvaţi problema fără a face uz de polimorfism şi comparaţi complexitatea unei astfel de implementări (incorecte din punct de vedere al programării orientate pe obiecte) cu cea bazată pe polimorfism. Apoi adăugaţi un nou fel de expresie în ambele rezolvări (de exemplu radical). Comparaţi dificultăţile întâmpinate în cele două variante.

7.7 Exerciţii

1. Creaţi o interfaţă cu trei metode. Implementaţi doar două metode din interfaţă într-o clasă. Va fi compilabilă clasa?
2. Se cere să se modeleze folosind obiecte şi interfeţe un sistem de comunicaţie prin mesaje între mai multe persoane. Pentru simplitate, considerăm că sunt disponibile doar două modalităţi de comunicare: prin e-mail sau prin scrisori (poştă clasică).

   O persoană X nu poate trimite un mesaj unei persoane Y direct, deoarece, de regulă, distanţele între persoane sunt prea mari. De aceea, pentru orice mesaj, persoana X trebuie să apeleze la un obiect transmiţător, care ştie să trimită mesaje către diverşi destinatari, în speţă, şi către persoana Y.

   Practic, dacă X doreşte să-i trimită un e-mail lui Y va apela la un obiect *EMailTransmitter*, iar dacă doreşte să-i trimită o scrisoare lui Y va opta pentru un obiect *MailTransmitter*. X va prezenta, în fiecare caz, identitatea proprie (vezi this), identitatea destinatarului, şi mesajul pe care doreşte să-l transmită.

   Evident, cele două moduri de comunicare trebuie să difere prin ceva, altfel nu s-ar justifica existenţa a două tipuri de obiecte. La transmiterea unui e-mail, destinatarul mesajului este notificat imediat de obiectul intermediar (*EMailTransmitter*). Notificarea unei persoane implică execuţia unei metode a clasei care modelează o persoană, fiind sarcina acestei metode de a cere obiectului *EMailTransmitter* în cauză să furnizeze mesajul respectiv. La transmiterea unei scrisori, destinatarul nu este notificat imediat. Chiar şi in realitate, o scrisoare nu este transmisă spre destinaţie imediat ce a fost depusă în cutia poştală, ci doar în anumite momente ale zilei

(dimineaţa, seara, etc). În aplicaţia noastră, considerăm că destinatarii scrisorilor sunt notificaţi doar atunci când cutia poştală *se umple* cu mesaje. Adică obiectul *MailTransmitter* care modelează sistemul clasic de poştă va avea un buffer de mesaje (de exemplu, un tablou) cu N elemente. Destinatarii mesajelor sunt notificaţi doar atunci când în buffer s-au adunat N mesaje. Notificarea unui destinatar presupune acelaşi lucru ca şi în cazul poştei electronice, pentru o tratare unitară a celor două cazuri. După ce toţi destinatarii au fost notificaţi şi şi-au *ridicat* scrisorile, buffer-ul va fi golit.

Sistemul trebuie să fie alcătuit din module slab cuplate (loosely coupled), ceea ce va duce la posibilitatea extinderii sale facile. Cu alte cuvinte, trebuie să ţinem cont că, în viitor, ar putea fi nevoie să folosim şi alte modalitaţi de comunicare prin mesaje (fax, SMS, etc.), care trebuie să poată fi integrate în sistemul nostru fără a fi nevoie de prea multe modificări.

Cerinţe. Sugestii de implementare Odată ce sistemul este modelat, el trebuie să permită o abordare de forma celei de mai jos:

```
// 4 persoane
Person p1=new Person("Paul");
Person p2=new Person("Andreea");
Person p3=new Person("Ioana");
Person p4=new Person("Gabriel");

//cream sistemul de transmitere prin e-mail-uri
Transmitter email=new EmailTransmitter();

//cream sistemul de transmitere prin scrisori, cu un buffer de 2 scrisori
Transmitter mail=new MailTransmitter(2);

//p1 doreste sa trimita un e-mail catre p2
p1.setTransmitter(email);
p1.send(p2,"Scrie-i Ioanei sa-mi dea adresa ei de e-mail!");

//p2 trimite o scrisoare catre p3. Scrisoarea nu va ajunge imediat,
//deoarece deocamdata este singura in "cutia postala"
p2.setTransmitter(mail);
p2.send(p3,"Paul zice sa-i trimiti adresa ta de e-mail");

//p4 trimite o scrisoare catre p1. Fiind a doua scrisoare,
//buffer-ul de scrisori se va umple si ambele scrisori vor fi trimise
p4.setTransmitter(mail);
p4.send(p1,"Ce mai faci?");
```

```
//p3 a primit in acest moment scrisoarea de la p2 si poate raspunde
//prin e-mail lui p1
p3.setTransmitter(email);
p3.send(p1,"Adresa mea de e-mail este: ioana@yahoo.com");
```

Transmitter se cere să fie *văzută* ca o interfaţă, sub forma:

```
public interface Transmitter
{
    public void store(Message message);
    public Message retrieve(Person receiver);
}
```

În codul anterior, *Message* este o clasă care modelează mesaje, adică conţine referinţe spre cele două persoane implicate în mesaj (sender-ul şi receiver-ul) şi conţine mesajul efectiv sub forma unui obiect String.

Clasele *EMailTransmitter* şi *MailTransmitter* vor implementa interfaţa *Transmitter*, adică trebuie să furnizeze implementări specifice pentru metodele *store* şi *retrieve*.

Fiecare persoană va conţine o referinţă spre obiectul *Transmitter* prin care vrea să transmită mesaje. Evident, acest obiect poate fi schimbat prin metoda *setTransmitter* pe care o are fiecare persoană.

Întreaga filosofie a aplicaţiei se rezumă la următoarele două reguli:

- în cadrul operaţiei *send*, se va folosi metoda *store* a obiectului *Transmitter* curent.
- când o persoană este notificată că a primit un mesaj, respectiva persoană va primi ca parametru al metodei de notificare şi o referinţă spre obiectul *Transmitter* care a făcut notificarea. Persoana destinatar va cere obiectului *Transmitter*, prin intermediul metodei *retrieve*, să-i furnizeze mesajul. Evident, trebuie să-şi prezinte identitatea, deoarece un transmiţător de scrisori va avea în buffer mai multe scrisori, şi trebuie s-o furnizeze pe aceea care corespunde destinatarului. După ce destinatarul a primit mesajul, el îl va afişa pe monitor.

Astfel, codul de mai sus va produce, de exemplu, textul:

- Paul said to Andreea (EMAIL) : "Scrie-i Ioanei sa-mi dea adresa ei de e-mail"
- Andreea said to Ioana (MAIL) : "Paul zice sa-i trimiti adresa ta de e-mail"
- Gabriel said to Paul (MAIL) : "Ce mai faci?"
- Ioana said to Paul (EMAIL) : "Adresa mea de e-mail este: ioana@yahoo.com"

3. Să se scrie un program Java care modelează activitatea unui ghişeu bancar. Sistemul este format din următoarele entităţi:
ContBancar cu următoarele atribute:

- numarCont(String)
- suma(float)

Client cu următoarele atribute:

- nume(String)
- adresa(String)
- conturi(tablou de elemente de tip ContBancar; un client trebuie să aibe cel puţin un cont, dar nu mai mult de 5)

Conturile bancare pot fi de mai multe feluri: în LEI şi în EURO. Conturile în EURO şi numai ele au o dobândă fixă, 0.3 EURO pe zi, dacă suma depăşeşte 500 EURO sau 0 în caz contrar, deci acest tip de cont trebuie să ofere serviciul *public float getDobanda()*. Pot exista transferuri între conturile în LEI şi numai între ele, în sensul că un cont de acest tip trebuie să ofere serviciul *public void transfer(ContBancar contDestinatie, float suma)*. Toate conturile implementează o interfaţă *SumaTotala* care are o metodă *public float getSumaTotala()*. Pentru conturile în lei suma totală este chiar suma existentă în cont iar pentru conturile în EURO este suma*36.000.
Banca cu următoarele atribute:

- clienti(tablou de elemente de tip Client)
- codBanca(String)

Conturile, pe lângă implementarea interfeţei *SumaTotala*, vor avea metode pentru setarea respectiv citirea atributelor ca unică modalitate de modificare (din exterior) a conţinutului unui obiect de acest tip precum şi metodele `public float getDobanda()`, *void transfer(ContBancar contDestinatie, float suma)* dar numai acolo unde este cazul.
Clasa *Client* va conţine un set de metode pentru setarea respectiv citirea atributelor ca unică modalitate de modificare (din exterior) a conţinutului unui obiect Client, un constructor prin intermediul căruia se vor putea seta numele, adresa clientului precum şi conturile deţinute de acesta; clasa trebuie să ofere şi o metodă pentru afişare.
Clasa *Banca* va implementa metode pentru efectuarea următoarelor operaţii, în contextul în care nu pot exista mai mulţi clienţi cu acelaşi nume.

- adăugarea unui client nou *public void add(Client c)*
- afişarea informaţiilor despre un client al cărui nume se transmite ca parametru *public void afisareClient(String nume)* în următoarea formă:
  - nume adresa
  - pentru fiecare cont deţinut, se va afişa doar suma totală pe o linie separată

În afara metodelor enumerate mai sus, clasele vor ascunde faţă de restul sistemului toate metodele şi atributele conţinute.

4. Fie o clasă *Project* care modelează un proiect software. Proiectele vor avea neapărat un manager. La un proiect se pot adăuga oricând participanţi, folosind metoda *public void addMember(Member m)*. Orice proiect are un titlu (String), un obiectiv (String) şi nişte (unul sau mai multe, vezi mai jos) fonduri (long). Managerul şi toţi participanţii vor fi programatori care au o vârstă (int) şi un nume (String). Un programator poate participa în mai multe proiecte.

   Există trei tipuri de proiecte: comerciale, militare şi open-source. Cele comerciale şi militare au un dead-line (String) şi un număr de maxim 15 membri, cele open-source au un mailing-list (String) şi număr nelimitat de membri. Cele militare au şi o parola (String), iar cele comerciale au fonduri de marketing (long) egale cu jumătate din fondurile normale şi un număr de echipe (int) mai mic decât numărul de membri.

   Toate proiectele implementează o interfaţă *Risky* care are o metodă *public double getRisk()*. Această metodă calculează riscurile legate de un proiect.
   La cele militare, riscul este numărul membrilor / lungimea parolei / fonduri.
   La cele comerciale, riscul este numărul echipelor * 3 / numărul membrilor / fonduri - fonduri de marketing.
   La proiectele open-source, riscul este numărul membrilor / fonduri.

   Clasa *InvestmentCompany* va modela o firmă care, date fiind niste proiecte în număr nelimitat, calculează care este proiectul cel mai puţin riscant. Va pune deci la dispoziţie o metodă *public void addProject(Project p)* şi o metodă *public Project getBestInvestment()*. Clasa *InvestmentCompany* va avea şi o metodă *public static void main(String[] args)* care să exemplifice folosirea clasei.

   Cerinţe:
   - Specificaţi clasele de care aveţi nevoie şi desenaţi ierarhia de clase dacă este cazul. Implementaţi clasele.
   - Unde (în ce clase) aţi ales să implementaţi interfaţa *Risky* şi de ce aţi făcut această alegere?

Bibliografie

1. Bruce Eckel. *Thinking in Java, 4th Edition*. Prentice-Hall, 2006. Capitolul Interfaces.
2. Martin Fowler. *UML Distilled, 3rd Edition*. Addison-Wesley, 2003.
3. Carmen De Sabata, Ciprian Chirilă, Călin Jebelean, *Laboratorul de Programare Orientată pe Obiecte*, Lucrarea 7 - Interfeţe - Aplicaţii, UPT 2002, variantă electronică.