Laborator 4:

Continuare Programare Orientată pe Obiecte

Întocmit de: Adina Neculai

Îndrumător: Asist. Drd. Gabriel Danciu

29 octombrie 2011

I. NOŢIUNI TEORETICE

A. Suprascrierea metodelor

O clasă derivată poate declara o metodă cu aceeaşi semnătură cu a unei metode din clasa de bază. Metoda din clasa derivată substituie astfel metoda din clasa de bază. Această tehnică se numeşte *suprascriere*. Cu alte cuvinte, la apelul metodei suprascrise din clasa derivată se va executa metoda declarată în clasa derivată. Dacă se doreşte apelul metodei ce aparţine clasei de bază, atunci în metoda din clasa descendentă se foloseşte cuvântul cheie *super* urmat de operatorul "." şi numele metodei suprascrise.

Sunt cazuri în care suprascrierea unei metode nu este dorită. Acest lucru se poate realiza prin adăugarea cuvântului cheie *final*. Astfel că, o metodă declarată *final* în superclasă nu poate fi suprascrisă în subclasă.

Diferenţa între o metodă suprascrisă şi una supraîncărcată este că în cazul celei suprascrise semnătura este identică(atât numărul parametrilor cât şi tipul acestora).

B. Clasa Object

În Java, toate clasele formează o anumită structură arborescentă, care are ca rădăcină clasa *Object*. Implicit, aceasta este superclasa tuturor claselor şi nu are o clasă părinte.

Metodele principale ale clasei Object sunt:

- *clone*

  Semnătura aceste metode este: **protected Object clone()** şi returnează o *clonă* a obiectului căruia i se aplică, deci un obiect cu aceeaşi stare şi acelaşi comportament. Cu toate acestea, obiectul rezultat nu este tocmai identic deoarece adresa de memorie către care face referinţă este diferită de cea a obiectului original.

  După ce se execută instrucţiunea *b=a.clone()*, în care a şi b sunt referinţe la obiecte, expresia *a==b* va avea valoarea *false*, deoarece valorile variabilelor referinţă a si b sunt diferite, în timp ce expresia *a.equals(b)* va întoarce valoarea *true*, deoarece cele două obiecte comparate au conţinuturi identice.

- *equals*

  Semnătura metodei este **public boolean equals(Object obj)** şi returnează valoarea *true* dacă şi numai dacă obiectele comparate sunt într-adevăr identice; adică obiectele au acelaşi conţinut şi aceeaşi adresă de memorie.

- *hashCode*

  Semnătura metodei este **public int hashCode()** şi returnează un întreg care este folosit de Java pentru a diferenţia două obiecte. Se recomandă ca atunci când se doreşte suprascrierea metodei *equals()*, să se suprascrie şi metoda *hashcode()*. Este important ca două obiecte egale conform metodei *equals()* să aibă aceleaşi hashcode-uri.

- *toString*

  Semnătura metodei este **public String toString()** şi returnează un şir de caractere de forma *java.lang.Object@hashCode* în care *hashCode* este exprimat în hexazecimal. Acest şir de caractere reprezintă obiectul căruia i se aplică. De aceea, se recomandă suprascrierea metodei *toString()* în toate subclasele clasei Object pentru ca mesajul să poată fi înţeles cu mai multă uşurinţă.

Pentru mai multe informaţii se recomandă citirea API-ului.

C. Clase şi metode abstracte

Forma generală a unei clase abstracte este:

[modificator_acces] abstract class NumeClasa{

}

O clasă *abstractă* este utilizată pentru modelarea unor concepte abstracte şi poate reprezenta doar clasa de bază în procesul de moştenire. Un lucru important de reţinut atunci când se lucrează cu clase abstracte este acela că o clasă abstractă nu poate fi instanţiată.

Alte caracterisitici ale unor clase de acest fel sunt următoarele:

- pot implementa constructori;

- pot avea atribute ce suportă modificări;

- pot avea metode abstracte. O *metodă abstractă* este o metodă ce nu are corp (instrucţiuni) şi care într-o clasă descendentă a clasei abstracte trebuie implementată. Dacă metoda declarată abstractă în clasa de bază nu este implementată în subclasă, atunci trebuie declarată din nou abstractă. Astfel şi subclasa devine una abstractă.

- pot avea şi metode clasice (corpul acestora este implementat în clasa abstractă);

- deşi clasele abstracte nu pot fi instanţiate, se pot declara obiecte de tipul claselor abstracte şi să fie instanţiate de clasele derivate.

D. Interfeţe

Forma generală a unei interfeţe este:

[modificator_acces] interface NumeInterfata [extends lista_interfete] {

}

O interfaţă este un fel de clasă abstractă cu diferenţa că într-o interfaţă nici o metodă declarată nu are voie să aibă corp. Interfeţele sunt utilizate pentru a separa implementarea de definiţia metodelor.

Spre deosebire de o clasă abstractă, o interfaţă nu poate defini un constructor, iar atributele existente sunt implicit declarate ca fiind statice şi finale, acestea devenind constante. Iar spre deosebire de o clasă obişnuită, o interfaţă poate moşteni mai multe interfeţe (intervine ideea de moştenire multiplă).

Forma generală a unei clase care moşteneşte o interfaţă sau o listă de interfeţe este:

[modificator_acces] class NumeClasa [extends NumeSuperclasa] implements lista_interfete{

}

De aici se poate trage concluzia că o clasă poate implementa una sau mai multe interfeţe.

De asemenea, se păstrează regula de la clasele abstracte cum că o clasă care moşteneşte o interfaţă trebuie să îi implementeze toate metodele. Dacă acest lucru un se întâmplă, atunci în clasă metodele devin abstracte.

II. PREZENTAREA LUCRĂRII DE LABORATOR

A. Suprascrierea metodelor

Se dă clasa *Superclasa* care conţine o singură metodă. Clasa *Subclasa* moşteneşte *Superclasa* şi suprascrie metoda *afiseazaDescriere*. Observaţi semnăturile metodelor; sunt identice.

```
1 public class Superclasa {
2    public void afiseazaDescriere() {
3       System.out.println("apelare afiseazaDescriere din superclasa...");
4    }
5 }
```

```
1 public class Subclasa extends Superclasa{
2    public void afiseazaDescriere(){
3       //apelare explicita a metodei din clasa de baza
4       //super.afiseazaDescriere();
5       System.out.println("apelare afiseazaDescriere din subclasa...");
6    }
7 }
```

În clasa descendentă, în metoda suprascrisă se poate apela metoda originală (din clasa de bază) folosind cuvântul cheie *super*. Decomentaţi linia de cod numărul 4 din *Subclasa* şi observaţi ce se afişează la următoarea rulare.

```
1  public class TestSuprascriere {
2     public static void main(String[] args){
3        Superclasa superClasa = new Superclasa();
4        Subclasa subClasa = new Subclasa();
5        superClasa.afiseazaDescriere(); System.out.println();
6        subClasa.afiseazaDescriere(); System.out.println();
7        superClasa = subClasa;
8        superClasa.afiseazaDescriere();
9     }
10 }
```

B. Clasa Object

Clasa *Point* are două atribute de tipul double şi suprascrie metodele *clone, equals, hashCode* şi *toString* ale clasei *Object*. Observaţi în linia de cod 1 cuvintele *implements Cloneable*. Orice clasă care extinde clasa Object şi care îi suprascrie metodele trebuie să implementeze interfaţa *Cloneable*. Despre interfeţe se discută pe larg în secţiunile I D şi II D.

Pentru a respecta proprietăţile metodei *hasCode*, de a returna un număr întreg care să diferenţieze două obiecte diferite, se implementează un algoritm care produce un astfel de număr. În

exemplul prezentat (linia de cod 30), algoritmul constă în înmulţirea cu un număr prim şi însumarea membrilor obiectului. Cu cât algoritmul matematic este mai complex cu atât regulile hashCode sunt respectate în mai multe cazuri.

```
1  public class Point implements Cloneable{
2     private double x,y;
3
4     public void setX(double x) {
5        this.x = x;
6     }
7
8     public void setY(double y) {
9        this.y = y;
10    }
11
12    protected Point clone(){
13       Point pObj = new Point();
14       pObj.x = this.x;
15       pObj.y = this.y;
16       return pObj; //obiectul pObj are membrii identici cu cei ai obiectului curent
17    }
18
19    public boolean equals(Object obj){
20       if (obj == this)
21          return true; //referinte egale
22       if (!(obj instanceof Point))
23          return false; //obj nu este de tipul Point
24       Point p = (Point) obj; //se face un downcasting, adica obj este convertit la tipul Point
25       if (this.x != p.x || this.y != p.y)
26          return false; //coordonata x sau y este diferita
27       return true;
28    }
29
30    public int hashCode(){
31       double result = 17;
32       result = 37*result + this.x;
33       result = 37*result + this.y;
34       return (int)result; /*se face un cast la tipul int, intrucat variabila result este una
35                de tip double, iar funtia hashCode returneaza o valoare de tip int*/
36    }
37
38    public String toString(){
39       return "x= "+x+"; y= "+y;
40    }
41 }
```

Clasa *TestObject* creează un obiect de tipul Point, îi setează proprietăţile şi îi apelează metodele.

```
1  public class TestObject {
2     public static void main(String[] args){
3        Point p1 = new Point();
4        p1.setX(3);
5        p1.setY(4);
6        Point p1Clone = p1.clone();
7        System.out.println("P1: "+p1.toString());
8        System.out.println("Clona lui P1: "+p1Clone.toString());
9        if (p1.equals(p1Clone))
```

```
10            System.out.println("Obiectele sunt identice!");
11        else System.out.println("Obiectele nu sunt identice!");
12        System.out.println("Hashcode-ul lui P1 este: "+p1.hashCode());
13        System.out.println("Hashcode-ul clonei lui P1 este: "+p1Clone.hashCode());
14    }
15 }
```

C. Clase şi metode abstracte

Pentru a exemplifica utilizarea claselor şi metodelor abstracte vom folosi clasa *Produs*. Să presupunem că vrem să derivăm un număr mai mare de clase din *Produs*: Carte, RamaFoto, etc. Se observă că acestea au în comun preţul, o descriere şi eventual o reducere a preţului în funcţie de anumite proprietăţi ale fiecărui produs în parte.

În acest caz, obiecte de genul *Carte* sau *RamaFoto* trebuie să facă parte din clase care moştenesc o superclasă comună, *Produs*. Mai mult, ar trebui ca Produs să implementeze metodele *afiseazaDescriere* şi *calculeazaReducere*. Pentru că implementarea metodelor depinde de tipul obiectului şi proprietăţilor acestuia, metodele se declară abstracte în clasa Produs.

```
1  public abstract class Produs {
2     private double pretUnitar;
3
4     Produs(double pretUnitar){
5        this.pretUnitar = pretUnitar;
6     }
7
8     public double getPretUnitar() {
9        return pretUnitar;
10    }
11
12    //metodele abstracte nu au corp
13    //ele vor fi implementate in clasele copil ale clasei Produs
14    public abstract void afiseazaDescriere();
15    public abstract double calculeazaReducere(int procent);
16 }
```

Clasa *Carte* extinde clasa *Produs* şi implementează metodele abstracte ale acesteia în modul ei propriu. La fel se întâmplă şi pentru clasa *RamaFoto*.

```
1  public class Carte extends Produs{
2     private String titlu, autor;
3
4     public Carte(double pretUnitar, String titlu, String autor) {
5        super(pretUnitar);
6        this.titlu = titlu;
7        this.autor = autor;
8     }
9
```

```
    //metoda afiseazaDescriere este implementata in sublclasa
    public void afiseazaDescriere(){
        System.out.println("Titlu cartii este: "+this.titlu+", iar autorul este: "+this.autor);
    }

    //metoda calculeazaReducere este implementata in sublclasa
    public double calculeazaReducere(int procent){
        System.out.println("Calculeaza reducere din Carte...");
        return 0.0;
    }
}
```

```
public class RamaFoto extends Produs{
    private int lungime, latime;

    public RamaFoto(double pretUnitar, int lungime, int latime) {
        super(pretUnitar);
        this.lungime = lungime;
        this.latime = latime;
    }

    //metoda afiseazaDescriere este implementata in sublclasa
    public void afiseazaDescriere(){
        System.out.println("Lungimea ramei este: "+this.lungime+", iar latimea este: "+this.latime);
    }

    //metoda calculeazaReducere este implementata in sublclasa
    public double calculeazaReducere(int procent){
        System.out.println("Calculeaza reducere in RamaFoto in functie de latime si lungime...");
        return 0.0;
    }
}
```

Clasa *TestProdus* creează un vector de obiecte de tipul Produs şi calculează preţul total al tuturor produselor declarate. Cum un obiect dintr-o clasă abstractă nu poate fi instanţiat cu tipul clasei abstracte, se instanţiază cu tipul subclaselor, în acest caz *Carte* şi *RamaFoto*.

```
public class TestProdus {
    public static void main(String[] args) {
        Produs[] listaProduse = new Produs[4];
        for(int i=0; i<listaProduse.length; i++){
            if (i % 2 == 0){ //daca i este par atunci se creeaza un obiect de tipul Carte
                listaProduse[i] = new Carte(12.3, "Fundatia "+i, "Isaac Asimov");
            }else{ //daca i este impar atunci se creeaza un obiect de tipul RamaFoto
                listaProduse[i] = new RamaFoto(5.5, 20-i, 15-i);
            }
        }
        double pretTotal = 0.0;
        for (int i=0; i<listaProduse.length; i++){
            //putem avea acces la metoda getPretUnitar() pentru ca aceasta se afla in clasa parinte.
            double pretProdusOriginal = listaProduse[i].getPretUnitar();
            double reducere = listaProduse[i].calculeazaReducere(i);
            pretTotal += pretProdusOriginal - reducere;
            listaProduse[i].afiseazaDescriere();
        }
        System.out.println("Pretul total al produselor este: "+pretTotal);
```

```
20      }
21  }
```

D. Interfeţe

Avem interfaţa *IPersoana* în care se află definiţia unei metode.

```
1  public interface IPersoana {
2      public String returneazaNumePersoana();
3  }
```

Interfaţa *IStudent* moşteneşte interfaţa *IPersoana*, implicit şi metoda unică a acesteia.

```
1  public interface IStudent extends IPersoana{
2      public boolean verificaStudentIntegralist();
3  }
```

Clasa *Student* implementează interfaţa *IStudent*, deci implementează atât metodele acesteia, cât şi pe cele ale interfeţei pe care *IStudent* o moşteneşte.

```
1  public class Student implements IStudent{
2      String nume;
3      double medieNote;
4
5      public Student(String nume, double medieNote){
6          this.nume = nume;
7          this.medieNote = medieNote;
8      }
9
10     //se implementeaza metoda din IStudent
11     public boolean verificaStudentIntegralist() {
12         if (this.medieNote >= 5){
13             return true;
14         }
15         return false;
16     }
17
18     //se implementeaza metoda mostenita de IStudent din IPersoana
19     public String returneazaNumePersoana() {
20         return this.nume;
21     }
22 }
```

Clasa *TestStudent* creează un obiect de tipul *Student* şi verifică dacă studentul este integralist sau nu, afişând un mesaj corespunzător.

```
1  public class TestStudent {
2      public static void main(String[] args) {
3          Student student = new Student("Ionescu Radu", 6.3);
4          if (student.verificaStudentIntegralist())
5              System.out.println(student.returneazaNumePersoana()+" este integralist");
6          else
```

```
7             System.out.println(student.returneazaNumePersoana()+" nu este integralist");
8         }
9     }
```

III. TEMĂ

1. Folosind conceptul de clase şi metode abstracte creaţi o aplicaţie care calculează pentru un cub şi pentru o sfera:

   - aria totala;
   - volumul;
   - centrul de greutate.

   Pentru acestea veţi avea nevoie de:

   - cub (reprezentat de 8 puncte);
   - sferă (reprezentată de 2 puncte: centrul şi un punct de pe margine);
   - punct (reprezentat de cele 3 coordonate în spatiu: x, y şi z);
   - metoda care calculează distanţa dintre 2 puncte aflate în spaţiu. Aceasta este necesară pentru aflarea lungimii unei laturi.