Laborator 3:

Introducere în Programarea Orientată pe Obiecte

Întocmit de: Adina Neculai

Îndrumător: Asist. Drd. Gabriel Danciu

23 octombrie 2011

I. NOŢIUNI TEORETICE

A. Clase şi obiecte

Unul dintre conceptele care defineşte programarea orientată pe obiecte este *obiectul*. Obiectul este caracterizat prin *stare*, modelată de atributele unei clase, şi prin *comportament*, modelat de metodele unei clase.

Un alt concept este *clasa*. Clasa este o colecţie de obiecte care partajează aceeaşi lista de atribute informaţionale (de stare) şi comportamentale. Aşadar, aceasta reprezintă atât structura unui obiect, cât şi funcţionalitatea acestuia. Mai mult, în acest context se spune că obiectul este o instanţă a unei clase.

Forma generală a unei clase:

[lista_Modificatori] class NumeClasa[extends NumeClasaDeBaza][implements lista_Interfata]

{

corp clasa

}

Definiţia unei clase trebuie să conţină obligatoriu cuvântul cheie *class*, numele clasei şi corpul clasei prins între acolade. Toţi ceilalţi termeni pot lipsi. Vom discuta despre aceşti termeni la momentul potrivit.

B. Supraîncărcarea metodelor

Forma generală a unei metode:

[lista_Modificatori] tip_de_returnat numeMetoda([lista_parametrii])

{

corp metoda

}

O metodă se defineşte prin semnătură şi prin corp. *Semnătura* constă în tipul returnat, numele metodei şi lista de parametri, pe când *corpul* metodei este reprezentat de instrucţiuni.

În Java, în aceeaşi clasă, se pot defini metode cu acelaşi nume, dar cu semnături diferite. Diferenţa poate consta în numărul de parametri, în tipul de date al acestora sau în ambele. Acest

proces se numeşte *supraîncărcare*. Este important de reţinut că Java nu ia în considerare tipul valorii returnate pentru a face diferenţierea metodelor supraîncărcate.

Motivul pentru care se folosesc acest gen de metode este că se elimină nevoia de a defini metode complet diferite care să facă în principiu acelaşi lucru. De asemenea, supraîncărcarea face posibilă comportarea diferită a metodelor în funcţie de argumentele primite.

C. Constructori

Într-o clasă, pe lângă atribute şi metode, se pot defini şi *constructori*. Aceştia sunt un tip de metodă specială cu următoarele caracteristici:

- numele lor trebuie să fie identic cu cel al clasei din care fac parte;
- nu au tip de returnat.

```
[lista_Modificatori] NumeClasa([lista_parametrii])
  {
     corp constructor
  }
```

Constructorii sunt utilizaţi pentru instanţierea unui obiect care face parte dintr-o anumită clasă şi sunt apelaţi în funcţie de variabilele pe care le primesc ca parametru. În cazul în care nu se declară niciun constructor(doar în acest caz) compilatorul creează un constructor implicit, având numele clasei, fără niciun parametru formal şi având corpul constructorului fără instrucţiuni.

Metodele constructor pot fi şi ele supraîncărcate, la fel ca metodele obişnuite, pentru a crea un obiect care are proprietăţi specifice în funcţie de argumentele transmise prin operatorul *new*.

D. Moştenire

Unul dintre principiile programării orientate pe obiecte este *moştenirea*. Acest principiu se bazează pe extinderea comportamentului unei clase existente prin definirea unei clase noi care moşteneşte conţinutul primei clase la care se adaugă noi proprietăţi şi funcţionalităţi.

Clasa existentă, care va fi moştenită, se numeşte clasa de bază, clasă părinte sau superclasă.

Clasa care realizează extinderea se numeşte subclasă, clasă derivată, clasă descendentă sau clasă copil. Cuvântul predefinit care se utilizează pentru moştenire este *extends*.

[lista_Modificatori] class NumeSubclasa [extends NumeClasaDeBaza]

```
{
   corp subclasa
}
```

În ce priveşte tipul de moştenire, în Java este permisă doar cea simplă. Adică, o subclasă poate extinde o singură superclasă. În aceeaşi ordine de idei, o superclasă poate fi moştenită de mai multe subclase diferite. Moştenirea multiplă nu este permisă în Java, dar este simulată cu ajutorul interfeţelor. Acest aspect îl vom trata ulterior.

Prin operaţia de derivare, constructorii clasei de bază nu se moştenesc, în schimb fiecare constructor al clasei derivate apelează un constructor al clasei de bază. Acest apel se poate face implicit(adăugat de compilator) sau explicit. Apelul explicit se face prin intermediul cuvântului cheie *super*. În plus, constructorii se apelează în ordinea derivării, de la superclasă la sublcasă.

```
public class NumeSubclasa extends NumeClasaDeBaza {
   NumeSubclasa(){
      super();
      //alte instructiuni constructor
   }
}
```

II. PREZENTAREA LUCRĂRII DE LABORATOR

A. Supraîncărcarea metodelor şi constructorilor

Codul sursă următor exemplifică conceptul de clasă şi obiect, împreună cu ideea de supraîncăr-care atât a constructorilor unei clase, cât şi a metodelor unei clase. Clasa *Point* are ca atribute coordonatele in plan ale unui punct şi calculează distanţa între două puncte. Metoda care descrie funcţionalitatea clasei Point (*computeDistance()*) este supraîncărcată. Acest lucru s-a realizat prin schimbarea numărului de parametri ai acesteia şi tipul lor.(observaţi linia de cod 29)

```java
public class Point {
   public double x, y;

   public Point() {// constructor fara parametri
      // this este un cuvant cheie ce reprezinta obiectul curent al clasei, intrucat nu se stie cum se numeste obiectul
      this.x = 0;
      this.y = 0;
   }

   //constructor cu 2 parametri; constructorul a fost supraincarcat
   public Point(double x, double y) {
      this.x = x;
      this.y = y;
   }

   // constructor cu un parametru; constructorul a fost supraincarcat
   public Point(double x) {
      this.x = x;
      this.y = 0;
   }

   // metoda calculeaza distanta intre 2 puncte aflate in plan
   public static double computeDistance(Point p1, Point p2){
   // se folosesc metode ale clasei Math din pachetul java.lang
      return Math.sqrt(Math.pow(p1.x - p2.x, 2) + Math.pow(p1.y - p2.y, 2));
   }

   // metoda este supraincarcata
   public static double computeDistance(double x1, double y1, double x2, double y2){
      return Math.sqrt(Math.pow(x1 - x2, 2) + Math.pow(y1 - y2, 2));
   }

   public static double computeDistance(Point p1){/** metoda este supraincarcata*/
      return Math.sqrt(Math.pow(p1.x, 2) + Math.pow(p1.y, 2));
   }

   public String toString(){
      return "x= "+x+"; y= "+y;
   }
}
```

Următoarea clasă creează obiecte de tipul *Point*. Aceste obiecte sunt instanţiate în moduri diferite. De asemenea, se apelează metoda statică *computeDistance()* cu parametri diferiţi.

```java
public class TestPoint {
   public static void main(String[] args) {
      Point p1 = new Point();
      System.out.println("Instatiere obiect prin constructor fara parametri: P1: "+p1.toString());
      Point p2 = new Point(2, 2);
      System.out.println("Instatiere obiect prin constructor cu 2 parametri: P2: "+p2.toString());
      Point p3 = new Point(3);
      System.out.println("Instatiere obiect prin constructor cu 1 parametru: P3: "+p3.toString());

      System.out.println("Calculeaza distanta intre P2 si P3: "+Point.computeDistance(p2, p3));
      System.out.println("Calculeaza distanta intre P2 si P3: "+Point.computeDistance(p2.x, p2.y, p3.x, p3.y));
```

```
12        System.out.println("Calculeaza distanta intre P(0,0) si P2: "+Point.computeDistance(p2));
13    }
14 }
```

B. Moştenire

Pentru a exemplifica principiul moştenirii se dă clasa de bază *Fruct*.

```
1  public class Fruct {
2     private String tipFruct;
3
4     public Fruct(String tipFruct) {
5        this.tipFruct = tipFruct;
6        System.out.println("Constructor Fruct...");
7     }
8
9     //metoda getTipFruct este una de tip final, adica aceasta nu mai poate fi modificata de clasele care o
          mostenesc
10    final String getTipFruct(){
11       return this.tipFruct;
12    }
13 }
```

Clasa *Para* este subclasă a clasei *Fruct*. Para este un fruct, aşadar se respectă principiul moştenirii. În clasa *Para* se adaugă noi atribute informaţionale (*greutate, culoare*) şi un nou comportament reprezentat de metodele *getGreutate(), getCuloare()*.

```
1  public class Para extends Fruct{
2     //atributele sunt private, deci domeniul lor de vizibilitate se afla doar in cadrul clasei Para
3     private double greutate;
4     private String culoare;
5
6     public Para(String tipFruct, double greutate, String forma) {
7        super(tipFruct); //apelare explicita a constructorului clasei de baza cu parametrul tipFruct
8        System.out.println("Constructor Para...");
9        this.greutate = greutate;
10       this.culoare = forma;
11    }
12
13    //metodele sunt publice si deci pot fi vazute si in afara clasei, spre deosebire de atribute
14    public double getGreutate() {
15       return greutate;
16    }
17
18    public String getCuloare() {
19       return culoare;
20    }
21 }
```

Clasa *TestFruct* instanţiază un obiect de tip *Para*, implicit de tip *Fruct*. De aceea avem acces şi la metoda *getTipFruct()* care nu aparţine propriu-zis clasei *Para*.

```
public class TestFruct {
   public static void main(String[] args){
      Para obPara = new Para("para", 3, "galben"); //referinta de tipul para
      System.out.println("Tip Fruct: "+obPara.getTipFruct()); //apelare metoda din clasa de baza
      System.out.println("greutate: "+obPara.getGreutate()); //apelare metoda din sublclasa
      System.out.println("culoare: "+obPara.getCuloare());
   }
}
```

Observaţi în urma rulării aplicaţiei ordinea în care se apelează constructorii.

III. TEMĂ

1. Să se construiască o clasă *Fractie* care să implementeze operaţiile de adunare, scădere, înmulţire, împărţire şi simplificare. Să se folosească această clasă într-un program.

2. Să se construiască o clasă *Forma2D* şi o clasă *Cerc* care moşteneşte clasa *Forma2D*. Să se afişeze lungimea cercului şi suprafaţa acestuia.