Curs 3

Programare Orientată pe Obiecte în limbajul Java

Programare Orientată pe Obiecte

Obiecte şi clase

- Obiecte şi clase
- Obiecte
- Clase
- Constructori
- Variabile şi metode membre
- Variabile şi metode de clasă
- Clasa Object
- Conversii automate între tipuri
- Tipul enumerare
- Clase imbricate
- Clase şi metode abstracte

Definire clasa Rectangle

```
class Rectangle {

    int x, y, width, height;
    Point  origin; // originea – x si y
    Dimension d; //dimensiunea – latime si inaltime

    public Rectangle() { … };
    public Rectangle (int x, int y, int w, int h) { … };
    public Rectangle (Point p, Dimension d) { … };

    public void setLocation( int x, int  y) { … };
    public void setSize (int width, int height) { … };

}
```

Crearea obiectelor

- **Declararea**

  NumeClasa numeObiect;

- **Instanțierea: new**

  numeObiect = new NumeClasa();

- **Inițializarea**

  numeObiect = new NumeClasa([argumente]);

  Rectangle r1, r2;
  r1 = new Rectangle();
  r2 = new Rectangle(0, 0, 100, 200);

- **Obiecte anonime**

Rectangle patrat = new Rectangle(new Point(0,0),
new Dimension(100, 100));

- **Memoria nu este pre-alocată !**

  Rectangle patrat;
  patrat.x = 10; //Eroare

Folosirea obiectelor (1)

- **Aflarea unor informații**
- **Schimbarea stării**
- **Executarea unor acțiuni**
- **obiect.variabila**

```
Rectangle patrat = new Rectangle(0, 0, 10, 200);
System.out.println(patrat.width);
patrat.x = 10;
patrat.y = 20;
patrat.origin = new Point(10, 20);
```

- **obiect.metoda([parametri])**

```
Rectangle patrat = new Rectangle(0, 0, 10, 200);
patrat.setLocation(10, 20);
patrat.setSize(200, 300);
```

Folosirea obiectelor (2)

- **Metode de accesare:**
  setVariabila, getVariabila
  patrat.width = -100;
  patrat.setSize(-100, -200);
  // Metoda poate refuza schimbarea

```
class Patrat {
    private double latura=0;
    public double getLatura()
    {
        return latura;
    }
    public double setLatura(double latura) {
        this.latura = latura;
    }
}
```

Distrugerea obiectelor

- **Obiectele care nu mai sunt referite vor fi distruse automat.**

Referințele sunt distruse:

• natural

• explicit, prin atribuirea valorii null.

```
class Test {
    String a;
    void init() {
        a = new String("aa");
        String b = new String("bb");
    }
    void stop() {
        a = null;
    }
}
```

Garbage Collector

- **Procesul responsabil cu eliberarea memoriei**

System.gc

”Sugerează” JVM să elibereze memoria

Finalizarea

- **Metoda finalize este apelată automat înainte de eliminarea unui obiect din memorie.**

finalize ≠ destructor

Declararea claselor

```
[public][abstract][final] class NumeClasa
  [extends NumeSuperclasa]
  [implements Interfata1 [, Interfata2 ...]]
  {
    // Corpul clasei
  }
```

- Moştenire simplă

```
class B extends A {...}
    // A este superclasa clasei B
    // B este o subclasa a clasei A
class C extends A,B // Incorect !
```

- Object este rădăcina ierarhiei claselor Java.

Corpul unei clase

- **Variabile membre**
- **Constructori**
- **Metode membre**
- **Clase imbricate (interne)**

```
// C++
class A {
    void metoda1();
        int metoda2() { ... }
}
    A::metoda1() { ... }
// Java
class A {
    void metoda1(){ ... }
    void metoda2(){ ... }
}
```

Constructorii unei clase

```
class NumeClasa {
    [modificatori] NumeClasa([argumente]) {
        // Constructor
    }
}
```

- Dacă pentru o clasă nu este definit niciun constructor, compilatorul inițializează variabilele membru cu valorile lor implicite, în funcție de tip, astfel:
  - Tipurile de date numerice cu 0
  - Tipul char cu caracterul vid (‘’)
  - Variabilele referință cu null

Apel explicit constructori - this

- **this apelează explicit un constructor al clasei.**

```
class Dreptunghi {
    double x, y, w, h;
    Dreptunghi(double x1, double y1, double w1, double
      h1) {
      // Implementam doar constructorul cel mai general
      x=x1; y=y1; w=w1; h=h1;
      System.out.println("Instantiere dreptunghi");
    }
    Dreptunghi(double w1, double h1) {
      this(0, 0, w1, h1);
      // Apelam constructorul cu 4 argumente
    }
    Dreptunghi() {
      this(0, 0);
      // Apelam constructorul cu 2 argumente
    }
}
```

Apel explicit constructori - super

super apelează explicit un constructor al superclasei!

```
class Patrat extends Dreptunghi {
     Patrat(double x, double y, double d) {
          super(x, y, d, d);
     }
}
```

Atenție!

- **Apelul explicit al unui constructor nu poate apărea decât într-un alt constructor şi trebuie să fie prima instrucțiune din constructorul respectiv.**

this

this şi super: **Sunt folosite în general pentru a rezolva conflicte de nume prin referirea explicită a unei variabile sau metode membre.**

```
class A {
     int x;
     A() {
          this(0);
     }
     A(int x) {
        this.x = x;
     }
     void metoda() {
        x ++;
     }
}
```

super

```
class B extends A {
    B() {
        this(0);
    }
    B(int x) {
        super(x);
    }
    void metoda() {
        System.out.println(“metoda clasei B”);
        super.metoda();
    }
}
```

Constructorul implicit

```
class Dreptunghi {
    double x, y, w, h;
    // Nici un constructor
}
class Cerc {
    double x, y, r;
    // Constructor cu 3 argumente
    Cerc(double x, double y, double r) { ... };
}
...
Dreptunghi d = new Dreptunghi();
// Corect (a fost generat constructorul implicit)
Cerc c;
c = new Cerc();
// Eroare la compilare !
c = new Cerc(0, 0, 100);
// Varianta corectă
```

Lanțul de apeluri al constructorilor

- Cazul în care o clasă de bază este extinsă (moştenită) de către o clasă copil.
- Constructorii nu se moştenesc!
- Constructorii pot fi invocați implicit sau explicit (*super*)

- Ori de câte ori este creat un obiect din clasa copil, este invocat mai întâi constructorul (fără parametri) al clasei părinte.

- Acesta este lanțul implicit de apeluri al constructorilor.

Apelul în lanț (implicit) al constructorilor

```
class Demo{
   int  value1;
   int  value2;

   Demo(){
      value1 = 1;
      value2 = 2;
      System.out.println("Inside 1st Parent Constructor");
   }
   Demo ( int a ) {
      value1 = a;
      System.out.println("Inside 2nd Parent Constructor");
   }
  public void display(){
     System.out.println("Value1 === "+value1);
     System.out.println("Value2 === "+value2);
  }
}
```

Lanțul de apeluri al constructorilor

```
class DemoChild extends Demo{
    int value3;
    int value4;

   DemoChild() {
            value3 = 3;
           value4 = 4;
          System.out.println("Inside the Constructor of Child");
    }
    public void display(){
           System.out.println("Value1 === "+value1);
            System.out.println("Value2 === "+value2);
            System.out.println("Value3 === "+value3);
            System.out.println("Value4 === "+value4);
   }
  public static void main(String args[]){
          DemoChild d1 = new DemoChild();
          d1.display();
  }
}
```

Lanțul de apeluri al constructorilor

- În conformitate cu modul implicit de apelare a constructorilor, atunci când este creat un obiect din clasa DemoChild, este apelat mai întâi constructorul Demo() al clasei părinte şi apoi constructorul DemoChild() al clasei copil.

Output :

Inside 1st Parent Constructor

Inside the Constructor of Child

Value1 === 1

Value2 === 2

Value3 === 3

Value4 === 4

Lanțul de apeluri al constructorilor

- Ce se petrece dacă eliminăm constructorul fără parametrii din clasa Demo?
- Dar dacă îi eliminăm pe amândoi?

- Se observă că, de fapt, este supradefinit constructorul clasei Demo.
- Dacă vrem să apelăm constructorul Demo(int a) în loc de constructorul implicit Demo() atunci când este creat un obiect din clasa copil?
- Se va folosi cuvântul cheie **“super”** pentru a apela constructorii clasei părinte

Lanțul de apeluri al constructorilor

```
class DemoChild extends Demo{
    int value3;
    int value4;

   DemoChild(){
          super(5);
          value3 = 3;
          value4 = 4;
         System.out.println("Inside the Constructor of Child");
    }

     public void display(){      …   }

     public static void main(String args[]){
               DemoChild d1 = new DemoChild();
              d1.display();
      }
}
```

Output: ?

Lanțul de apeluri al constructorilor

Output:

Inside 2nd Parent Constructor

Inside the Constructor of Child

Value1 === 5

Value2 === 0

Value3 === 3

Value4 === 4

Lanțul de apeluri al constructorilor

```
class DemoChild extends Demo{
    int value3;
    int value4;
    DemoChild(){
          // super(5);
          value3 = 3;
          value4 = 4;
         System.out.println("Inside the 1st Constructor of Child");
    }
    DemoChild(int a){
          // this();
          value3 = a;
            System.out.println("Inside the 2nd Constructor of Child");
    }
    public void display() {…}

    public static void main(String args[]){
          DemoChild d2 = new DemoChild (11);
          d2.display();
      }
}
```

Lanțul de apeluri al constructorilor

Inside 1st Parent Constructor
Inside the 2nd Constructor of Child
Value1 === 1
Value2 === 2
Value3 === 11
Value4 === 0

// decomentare *this*
Inside 1st Parent Constructor
Inside the 1st Constructor of Child
Inside the 2nd Constructor of Child
Value1 === 1
Value2 === 2
Value3 === 11
Value4 === 4

// decomentare şi *super*
Inside 2nd Parent Constructor
Inside the 1st Constructor of Child
Inside the 2nd Constructor of Child
Value1 === 5
Value2 === 0
Value3 === 11
Value4 === 4

Modificatorii de acces

- ***Modificatorii de acces*** sunt cuvinte rezervate ce controlează accesul celorlalte clase la membrii unei clase.

Specificator	Clasa	Subcls*	Pachet	Oriunde
Private	X			
Implicit	X		X	
Protected	X	X	X	
Public	X	X	X	X

- dacă nu este specificat nici un modificator de acces, implicit nivelul de acces este la nivelul pachetului
- * subclasă din alt pachet

Declararea variabilelor

```
class NumeClasa {
    // Declararea variabilelor
    // Declararea metodelor
}
```

[modificatori] Tip numeVariabila [= valoare];

unde un modificator poate fi :

- **public, protected, private**
- **static, final, transient, volatile**

```
class Exemplu {
    double x;
    protected static int n;
    public String s = "abcd";
    private Point p = new Point(10, 10);
    final static long MAX = 100000L;
}
```

Final, transient, volatile

final:

```
class Test {
    final int MAX;
    Test() {
       MAX = 100; // Corect
       MAX = 200; // Eroare la compilare !
    }
}
```

transient:

- folosit la serializarea obiectelor, pentru a specifica ce variabile membre ale unui obiect nu participă la serializare.

volatile:

- folosit pentru a semnala compilatorului să nu execute anumite optimizări asupra membrilor unei clase.
- o facilitate avansată a limbajului Java.

Declararea metodelor

```
[modificatori] TipReturnat numeMetoda ([argumente])
    [throws TipExceptie1, TipExceptie2, ...]
{
    // Corpul metodei
}
```

unde un modificator poate fi :

- **public**, **protected**, **private**
- **static**, **abstract**, **final**, **native**, **synchronized**

```
class Student {
    ...
    final float calcMedie(float note[]) {
        ...
    }
}
class StudentInformatica extends Student {
    float calcMedie(float note[]) {
        return 10.00;
    }
}// Eroare la compilare !
```

Tipul returnat de o metodă

- return [valoare]
- void nu este implicit

```
public void afisareRezultat() {
        System.out.println("rezultat");
}
private void deseneaza(Shape s) {
        ...
        if …return;
        ….
}
```

- return trebuie să apară în toate situațiile, atunci când am tip returnat:

```
double radical(double x) {
        if (x >= 0)
                return Math.sqrt(x);
        else {
                System.out.println("Argument negativ !");
                // Eroare la compilare
                // Lipseste return pe aceasta ramura
        }
}
```

Argumentele metodelor

- Numele argumentelor primite trebuie să difere între ele şi nu trebuie să coincidă cu numele nici uneia din variabilele locale ale metodei.
- Pot să coincidă cu numele variabilelor membre ale clasei, caz în care diferențierea dintre ele se va face prin intermediul variabile *this*.

```
class Cerc {
    int x, y, raza;
    public Cerc(int x, int y, int raza) {
        this.x = x;
        this.y = y;
        this.raza = raza;
    }
}
```

Trimiterea parametrilor (1)

TipReturnat metoda([Tip1 arg1, Tip2 arg2, ...])

- **Argumentele sunt trimise doar prin valoare (pass-by-value).**

```
void metoda(StringBuffer sir, int numar) {
    // StringBuffer este tip referinta
    // int este tip primitiv
    sir.append("abc");
    numar = 123;
}
...
StringBuffer s=new StringBuffer();
int n=0;
metoda(s, n);
System.out.println(s + ", " + n);
// s va fi "abc", dar n va fi 0
```

Trimiterea parametrilor (2)

```
void metoda(String sir, int numar) {
	// String este tip referinta
	// int este tip primitiv
	sir = "abc";
	numar = 123;
}
...
String s=new String(); int n=0;
metoda(s, n);
System.out.println(s + ", " + n);
// s va fi "", n va fi 0

void schimba(int a, int b) {
	int aux = a;
	a = b;
	b = aux;
}
...
int a=1, b=2;
schimba(a, b);
// a va ramane 1, iar b va ramane 2
```

Trimiterea parametrilor (3)

```
class Pereche {
    public int a, b;
}
...

void schimba(Pereche p) {
    int aux = p.a;
    p.a = p.b;
    p.b = aux;
}
...

Pereche p = new Pereche();
p.a = 1, p.b = 2;
schimba(p);
//p.a va fi 2, p.b va fi 1
```

Metode cu număr variabil de argumente

[modif] TipReturnat metoda(TipArgumente ... args)

- Tipul argumentelor poate fi referință sau primitiv.

```
void metoda(Object ... args) {
    for(int i=0; i<args.length; i++)
        System.out.println(args[i]);
}

...

metoda("Hello");

metoda("Hello", "Java", 1.5);
```

Conversii automate între tipuri - autoboxing

byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

```
Integer obi = new Integer(1);
int i = obi.intValue();
Boolean obb = new Boolean(true);
boolean b = obb.booleanValue();
// Doar de la versiunea 1.5 !
Integer obi = 1;
int i = obi;
Boolean obb = true;
boolean b = obb;
```

Exemplu

```
public class Complex {
	private double a, b;
	...
	public Complex aduna(Complex comp) {
		return new Complex(a + comp.a, b + comp.b);
	}
	public boolean equals(Object obj) {
		if (obj == null) return false;
		if (!(obj instanceof Complex)) return false;
		Complex comp = (Complex) obj;
		return ( comp.a==a && comp.b==b);
	}
	public String toString() {
		if (b > 0) return a + ”+” + b + ”*i”;
		 return a + “” + b + ” *i";
	}
}
...
Complex c1 = new Complex(1,2);
Complex c2 = new Complex(2,3);
System.out.println(c1.aduna(c2));		// 3.0 + 5.0i
System.out.println(c1.equals(c2));		 // false
```