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

Obiecte şi clase

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

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
    }
}
```

- **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
    }
```

Apel explicit constructori

```
    Dreptunghi() {
        this(0, 0);
        // Apelam constructorul cu 2 argumente
    }
}
```

- **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() {
        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
```

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**			
Protected	**X**	**X***		
Public	**X**	**X**	**X**	**X**
Implicit	**X**		**X**	

- dacă nu este specificat nici un modificator de acces, implicit nivelul de acces este la nivelul pachetului

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) {
    ...
    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
    }
}
```

Tip - Subtip

- **int metoda() { return 1.2;} // Eroare**
- **int metoda() { return (int)1.2;} // Corect**
- **double metoda() {return (float)1;} // Corect**

Clasă - Subclasă

```
Poligon metoda1( ) {
        Poligon p = new Poligon();
        Patrat t = new Patrat();
        if (...)
                return p; // Corect
        else
                return t; // Corect
}
Patrat metoda2( ) {
        Poligon p = new Poligon();
        Patrat t = new Patrat();
        if (...)
                return p; // Eroare
        else
                return t; // Corect
}
```

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);
```

Polimorfism (1)

- Supraîncărcarea (overloading)
- Supradefinirea (overriding)

```
class A {
    void metoda() {
        System.out.println("A: metoda fara parametru");
    }
    // Supraîncărcare
    void metoda(int arg) {
        System.out.println("A: metoda cu un
          parametru");
    }
}
class B extends A {
    // Supradefinire
    void metoda() {
        System.out.println("B: metoda fara parametru");
    }
}
```

Polimorfism (2)

O metodă supradefinită poate :

• să ignore codul metodei părinte:

```
B b = new B();
b.metoda();
// Afişează "B: metoda fara parametru"
```

• să extindă codul metodei părinte:

```
class B extends A {
// Supradefinire prin extensie
    void metoda() {
        super.metoda();
        System.out.println("B: metoda fara parametru");
    }
}
. . .
B b = new B();
b.metoda();
/* Afişează ambele mesaje:
"A: metoda fara parametru"
"B: metoda fara parametru" */
```

- **În Java nu este posibilă supraîncărcarea operatorilor.**

Variabile de instanță şi variabile de clasă

```
class Exemplu {
   int x ; //variabila de instanta
}
```

- variabilă de instanță: la fiecare creare a unui obiect al clasei Exemplu sistemul alocă o zonă de memorie separată pentru memorarea valorii lui x.

```
class Exemplu {
   static int sx ; //variabila de clasă
}
```

- Pentru variabilele de clasă (statice) sistemul alocă o singură zonă de memorie la care au acces toate instanțele clasei respective, ceea ce înseamnă că dacă un obiect modifică valoarea unei variabile statice ea se va modifica şi pentru toate celelalte obiecte.

Variabile de clasă

- Deoarece nu depind de o anumită instanță a unei clase, variabilele statice pot fi referite şi sub forma:

  NumeClasa.numeVariabilaStatica

  Ex. Exemplu.sx

- Inițializarea variabilelor de clasă se face o singură dată, la încărcarea în memorie a clasei respective

```
class Exemplu {
    static final double PI = 3.14;
    static long nrInstante = 0;
    static Point p = new Point(0,0);
}
```

Variabile de instanță şi variabile de clasă

```
class Exemplu {
    int x ; // Variabila de instanta
    static long n; // Variabila de clasa
}
    ...
    Exemplu o1 = new Exemplu();
    Exemplu o2 = new Exemplu();
    o1.x = 100;
    o2.x = 200;
    System.out.println(o1.x); // Afiseaza 100
    System.out.println(o2.x); // Afiseaza 200
    o1.n = 100;
    System.out.println(o2.n); // Afiseaza 100
    o2.n = 200;
    System.out.println(o1.n); // Afiseaza 200
    System.out.println(Exemplu.n); // Afiseaza 200
    // o1.n, o2.n si Exemplu.n sunt referinte la aceeasi
        // valoare
```

Metode de instanță şi metode de clasă

- **metodele de instanță operează atât pe variabilele de instanță cât şi pe cele statice ale clasei;**
- **metodele de clasă operează doar pe variabilele statice ale clasei.**

```
class Exemplu {
    int x ;      // Variabilă de instanță
    static long n; // Variabilă de clasă
    void metodaDeInstanta() {
        n ++; // Corect
        x --; // Corect
    }
    static void metodaStatica() {
        n ++; // Corect
        x --; // Eroare la compilare !
    }
}
```

Metode de instanță şi metode de clasă

- Intocmai ca şi la variabilele statice, întrucât metodele de clasă nu depind de starea obiectelor clasei respective, apelul lor se poate face şi sub forma:

  NumeClasa.numeMetodaStatica

```
Exemplu.metodaStatica(); // Corect
Exemplu obj = new Exemplu();
obj.metodaStatica(); // Corect
```

- Metodele de instanță nu pot fi apelate decât pentru un obiect al clasei respective:

```
Exemplu.metodaDeInstanta(); // Eroare
Exemplu obj = new Exemplu();
obj.metodaDeInstanta(); // Corect
```

Utilitatea membrilor de clasă

- **folosiți pentru a pune la dispoziție valori şi metode independente de starea obiectelor dintr-o anumită clasă.**
- **Declararea eficientă a constantelor**

```
class Exemplu {
    static final double PI = 3.14;
    // Variabila finala de clasa
}
```

- **Numărarea obiectelor unei clase**

```
class Exemplu {
    static long nrInstante = 0;
    Exemplu() {
        // Constructorul este apelat la fiecare instantiere
        nrInstante ++;
    }
}
```

- **Implementarea funcțiilor globale**

Blocuri statice de inițializare

```
static {
    // Bloc static de initializare;
    ...
}
public class Test {
    // Declaratii de variabile statice
    static int x = 0, y, z;
    // Bloc static de initializare
    static {
        System.out.println("Initializam...");
        int t=1;
        y = 2;
        z = x + y + t;
    }
    Test() { ... }
    }
}
```

Blocuri statice de inițializare

- Variabilele statice ale unei clase sunt inițializate la un moment care precede prima utilizare activă a clasei respective.
- Momentul efectiv depinde de implementarea maşinii virtuale Java şi poartă numele de inițializarea clasei. În această etapă sunt executate şi blocurile statice de inițializare ale clasei.
- Variabilele referite într-un bloc static de inițializare trebuie să fie obligatoriu de clasă sau locale blocului.

Clasa Object

Object este superclasa tuturor claselor.

```
class Exemplu {}
class Exemplu extends Object {}
```

- clone
- equals
- finalize
- toString.

```
Exemplu obj = new Exemplu();
System.out.println("Obiect=" + obj);
//echivalent cu
System.out.println("Obiect=" + obj.toString());
```

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
```

Conversii automate între tipuri

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;
```