

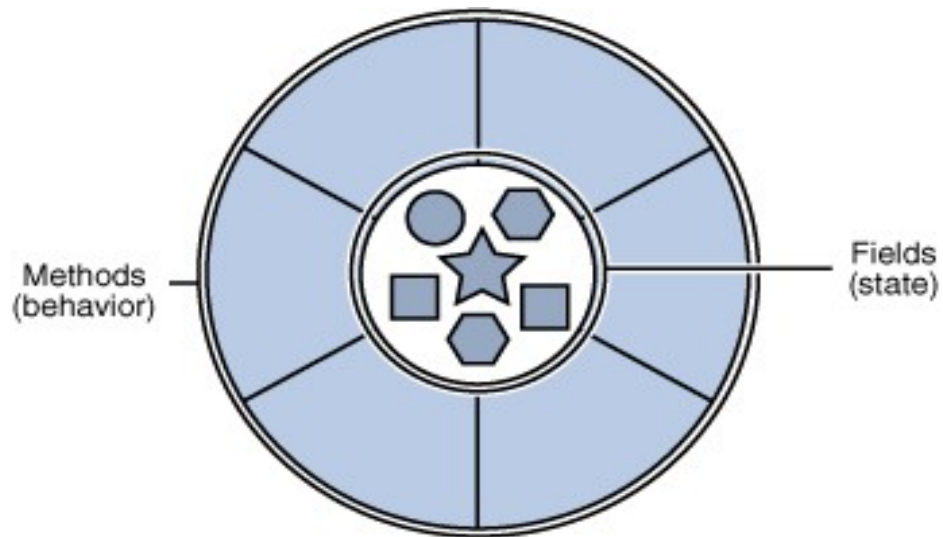
Obiecte și clase

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

Concepte POO

- **Obiect** = Entitate software descrisă de o *stare* și de un *comportament*.
- **Clasă** = Prototip ce descrie obiecte - obiectele sunt instanțe ale claselor.
- **Referință** = Entitate ce oferă informații necesare localizării în mod unic a unui obiect
- **Program** = Mulțime dinamică de obiecte ce interacționează
- **Interfață** = Contract la care aderă o anumită clasă
- **Pachet** = Spațiu de nume - necesar organizării claselor și interfețelor.

Ce este un obiect ?



- Identitate
- Stare
- Comportament

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-allocată !

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

Folosirea obiectelor

- Aflarea unor informații
- Schimbarea stării
- Executarea unor acțiuni

obiect.variabila

```
Rectangle patrat = new Rectangle(0, 0, 100, 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, 100, 200);  
patrat.setLocation(10, 20);  
patrat.setSize(200, 300);
```

Metode de accesare:
setVariabila, getVariabila

```
patrat.width = -100;  
patrat.setSize(-100, -200);  
// Metoda poate refuza schimbarea
```

```
class Patrat {  
    private double latura=0;  
  
    // accesori  
    public double getLatura() {  
        return latura;  
    }  
  
    // mutator  
    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` \neq `destructor`

Crearea claselor

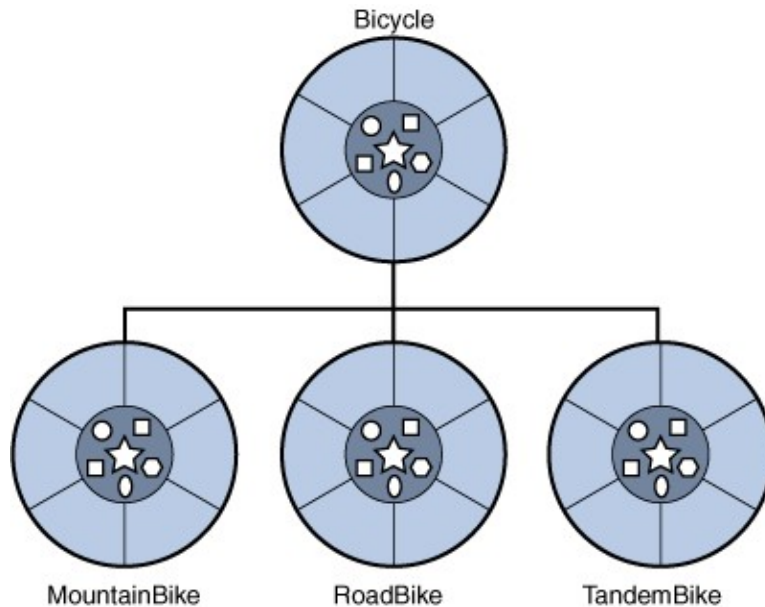
Clasă =

- Prototip ce descrie obiecte - obiectele sunt instanțe ale claselor.
- Construcție sintactică formată din entități (variabile, metode, etc.) membre.
- Tip de date

Declararea unei clase

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

Moșternirea



Moștenirea este doar 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
- Metodelor 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) {...}
    Dreptunghi(double w1, double h1) {
        this(0, 0, w1, h1);
    }
}
```

super apelează explicit un constructor al superclasei.

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

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 corecta
```

Modificatori de acces: `public`, `protected`,
`private` și cel implicit.

Declararea variabilelor

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

[modificatori] Tip numeVariabila [= valoare];

unde un modificador 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;  
}
```

this și super

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

Declararea metodelor

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

unde un modificador 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.

```
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
}
```

Trimiterea parametrilor

```
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
```

```
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
```

```
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)
```

```
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

polys=numero, morphe=forma

- Supraîncarcarea (overloading)
- Supradefinirea (overriding)

```
class A {
    void metoda() {
        System.out.println("A: metoda fara parametru");
    }
    // Supraincarcare
    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");
    }
}
```

O metodă supradefinită poate să:

- **ignore** codul metodei părinte:

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

- **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();
/* Afiseaza ambele mesaje:
"A: metoda fara parametru"
"B: metoda fara parametru" */
```

In Java nu este posibilă supraîncărcarea operatorilor.

Variabile de instanță și de clasă

```
class Exemplu {
    int x ;           // Variabila de instanta
    static long n;   // Variabila de clasa
}

...
Exemplu o1 = new Exemplu(); o1.x = 100;
Exemplu o2 = new Exemplu(); 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
```

Inițializarea

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

Metode de instanță și de clasă

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

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

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

Utilitatea membrilor de 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() { ... }  
}  
}
```

Clase imbricate

```
class ClasaDeAcoperire{
    class ClasaImbricata1 {
        // Clasa membru
        // Acces la membrii clasei de acoperire
    }
    void metoda() {
        class ClasaImbricata2 {
            // Clasa locala metodei
            // Acces la mebrii clasei de acoperire si
            // la variabilele finale ale metodei
        }
    }
}
```

Identificare claselor imbricate

```
ClasaDeAcoperire.class
ClasaDeAcoperire$ClasaImbricata1.class
ClasaDeAcoperire$ClasaImbricata2.class
```

Clase și metode abstracte

```
[public] abstract class ClasaAbstracta ... {  
    // Declaratii uzuale  
    // Declaratii de metode abstracte  
}
```

Metode abstracte

```
abstract class ClasaAbstracta {  
    abstract void metodaAbstracta(); // Corect  
    void metoda();                  // Eroare  
}
```

- O clasă abstractă poate să nu aibă nici o metodă abstractă.
- O metodă abstractă nu poate apărea decât într-o clasă abstractă.
- Orice clasă care are o metodă abstractă trebuie declarată ca fiind abstractă.

Exemple:

Number: Integer, Double, ...

Component: Button, List, ...

Clasa Object

`Object` este superclasa tuturor claselor.

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

Orice obiect, inclusiv tablourile, implementeaza metodele acestei clase.

- `toString`
- `equals`
- `hashCode`
- `getClass`
- `clone`
- `finalize`
- ...

toString

```
public String toString()
```

Returnează reprezentarea ca șir de caractere a unui obiect.

```
public class Object {  
    public String toString() {  
        return getClass().getName() + '@' +  
            Integer.toHexString(hashCode())  
    }  
    . . .  
}
```

Se recomandă ca orice clasă să supradefinească **toString!**

Invocare implicită:

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

equals

```
public boolean equals(Object obj)
```

Indică dacă obiectul specificat `obj` este
”egal” cu cel curent `this`.

Definește o *relație de echivalență*:

- *reflexivă*: `x.equals(x) == true`
- *simetrică*: `x.equals(y) == y.equals(x)`
- *tranzitivă*

În plus `x.equals(null)` trebuie să re-
turneze `false`

```
public class Object {  
    public boolean equals(Object obj) {  
        return (this == obj);  
    }  
    . . .  
}
```

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() {
        String semn = (b > 0 ? "+" : "-");
        return a + semn + 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
```

Clonarea obiectelor: `clone`

```
public Object clone()
```

Creează și returnează o ”copie” a obiectului curent `this`.

Clona trebuie să respecte proprietățile:

- `x.clone() != x`
- `x.clone().getClass() == x.getClass()`
- `x.clone().equals(x)`
- Independență

Implementarea implicită a metodei `clone` creează o *copie superficială* a obiectului clonat (*shallow copy*).

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; (auto)boxing  
int i = obi; //(auto)unboxing
```

```
Boolean obb = true;  
boolean b = obb;
```

Tipuri de date enumerare

enum

```
public class CuloriSemafor {
    public static final int ROSU = -1;
    public static final int GALBEN = 0;
    public static final int VERDE = 1;
}
...
// Exemplu de utilizare
if (semafor.culoare == CuloriSemafor.ROSU)
    semafor.culoare = CuloriSemafor.GALBEN);
...

public enum CuloriSemafor { ROSU, GALBEN, VERDE };
// Utilizarea structurii se face la fel
```