

Curs 2  
**Principiile Programării Orientate pe  
Obiecte**

Programare Orientată pe Obiecte



# Tehnici de programare

## Programarea procedurală

- I Modul în care este abordată programarea, din punct de vedere al descompunerii programelor
- I Paradigme

### **Programarea procedurală**

- I prima modalitate de programare, încă frecvent folosită
- I descompunerea programului în proceduri (funcții) care sunt apelate în ordinea de desfășurare a algoritmului.
- I limbajele care suportă această tehnică de programare prevăd posibilități de transfer a argumentelor către funcții și de returnare a valorilor rezultate.
- I limbajul Fortran: primul limbaj de programare procedurală .
- I au urmat Algol60, Algol68, Pascal, iar C este unul din ultimele invenții în acest domeniu.

## Programarea modulară (structurată)

- I accentul în proiectarea programelor s-a deplasat de la proiectarea procedurilor către organizarea datelor, datorită creșterii dimensiunii programelor.
- I stilul de programare este în continuare procedural, dar datele și procedurile sunt grupate în module. Gruparea de date și proceduri în module nu implică și o asociere strictă între acestea.
- I modul: o mulțime de proceduri corelate, împreună cu datele pe care le manevrează
- I tehnică de ascundere a datelor (*data-hiding*): posibilitatea de ascundere a unor informații definite într-un modul față de celelalte module.
- I Modularitatea și ascunderea informațiilor sunt caracteristici implicite în programarea orientată pe obiecte.

# Programarea orientată pe obiecte

- I programarea procedurală și structurată: descriere a algoritmilor ca o secvență de pași care duc de la datele inițiale la rezultatul căutat.
- I limbaje de programare orientate la o clasă concretă de probleme: sisteme de dirijare cu baze de date, modelare ș.a.
- I a apărut necesitatea sporirii siguranței programelor - interzicerea accesului neautorizat la date.
- I dezvoltarea sistemelor *orientate pe obiecte*, bazate pe *programarea orientată pe obiecte*, care a cunoscut o amploare deosebită în anii 90.
- I programarea orientată pe obiecte presupune:
  - determinarea și descrierea claselor utilizate în program
  - crearea exemplarelor de obiecte necesare
  - determinarea interacțiunii dintre ele.

# Programarea orientată pe obiecte

- I Reprezintă aplicarea în domeniul programării a unei metode din tehnică (tehnologia orientată pe obiecte, care se bazează pe modelul obiect)
- I Modelul obiect al unei aplicații implică patru principii importante și anume:
  - § abstractizare;
  - § încapsulare;
  - § modularitate;
  - § ierarhizare.
- I Primele aplicații: limbajul Simula, care a stat la baza Smalltalk, Object Pascal, C++, Clojure, Ada, Eiffel.
- I modelul obiect: un concept unificator în știința calculatoarelor, aplicabil nu numai în programare, ci și în arhitectura calculatoarelor, în proiectarea interfețelor utilizator, în baze de date.

# Programarea orientată pe obiecte

- I *Programarea orientată pe obiecte (object-oriented programming) este o metodă de programare în care programele sunt organizate ca și colecții de obiecte cooperante, fiecare dintre ele reprezentând o instanță a unei clase, iar clasele sunt membre ale unei ierarhii de clase, corelate între ele prin relații de moștenire.*

Există trei părți importante ale acestei definiții:

- I Se folosesc obiecte, nu algoritmi, ca unități constructive de bază , obiectele fiind componente ale unei ierarhii de agregare.
- I Fiecare obiect este o instanță (un exemplar) al unei clase.
- I Clasele sunt componente ale unei ierarhii de tip, fiind corelate între ele prin relații de moștenire.

Obs: Dacă lipsește una din aceste caracteristici, programarea nu se mai numește orientată pe obiecte, ci programare prin abstracționarea datelor, deoarece o clasă este un tip de date abstract.

# Limbaaj de programare orientată pe obiecte

## I Cerințe:

1. Suportă obiecte (instanțe ale unor clase), clasele fiind tipuri definite de utilizator (numite și tipuri abstracte de date).
2. Tipurile (clasele) pot moșteni attribute de la alte clase, numite clase de bază .

- I Dacă un limbaj nu suportă direct moștenirea între clase se numește limbaj de programare bazat pe obiecte (*object-based*), cum este, de exemplu, limbajul Ada.

## Principii POO: *Abstractizarea*

- I posibilitatea ca un program să ignore unele aspecte ale informației pe care o manipulează, adică posibilitatea de a se concentra asupra esențialului.
- I înseamnă identificarea similitudinilor între diferite entități, situații sau procese din lumea reală , concentrarea atenției asupra acestor aspecte comune și ignorarea pentru început a detaliilor.
- I identifică trăsăturile caracteristice esențiale ale unui obiect, care îl deosebesc de toate celelalte feluri de obiecte.
- I fiecare obiect în sistem are rolul unui “actor” abstract, care poate executa acțiuni, își poate modifica și comunica starea și poate comunica cu alte obiecte din sistem fără a dezvălui cum au fost implementate acele facilități.
- I procesele, funcțiile sau metodele pot fi de asemenea abstracte



# Principii POO:

## Încapsularea

- I ascunderea de informații (data-hiding)
- I obiectele nu pot schimba starea internă a altor obiecte în mod direct (ci doar prin metode puse la dispoziție de obiectul respectiv);
- I doar metodele proprii ale obiectului pot accesa starea acestuia.
- I procesul de compartimentare a elementelor unei abstractizări în două părți: structura și comportarea;
- I încapsularea separă comportarea (accesată prin interfață) de structură, definită prin implementare.
- I fiecare tip de obiect expune o interfață pentru celelalte obiecte care specifică modul cum acele obiecte pot interacționa cu el.

# Principii POO

## Modularizarea

---

- I este procesul de partiționare a unui program în componente individuale (module) ceea ce permite reducerea complexității programului prin definirea unor granițe bine stabilite și documentate în program.
  - modularizarea constă în **partiționarea programului în module** care pot fi compilate separat, dar care au conexiuni cu alte module ale programului.
  - modulele servesc ca și containere în care sunt declarate clasele și obiectele programului.

# Principii POO

## Ierarhizarea

- I modalitatea de a **ordona abstractizările** (tipurile abstracte de date).
- I ierarhiile pot să denote relații de tip sau relații de agregare.
  - **Relațiile de tip** sunt definite prin moștenirile între clase, prin care o clasă (clasa derivată) moștenește structura sau comportarea definită în altă clasă (clasa de bază ).
  - **Relațiile de agregare** specifică compunerea unui obiect din mai multe obiecte mai simple.  
Obs: în limbajele de programare procedurală agregarea se realiza prin structuri de tip înregistrare (record în Pascal, struct în C, etc).

# Principii POO

## Moștenirea

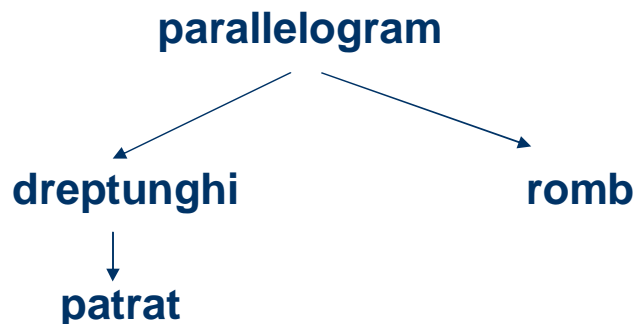
- I permite definirea și crearea unor clase specializate plecând de la clase (generale) care sunt deja definite
- I permite construirea unor clase noi, care păstrează caracteristicile și comportarea, deci datele și funcțiile membru, de la una sau mai multe clase definite anterior, numite **clase de bază**, fiind posibilă redefinirea sau adăugarea unor date și funcții noi.
- I o clasă ce moștenește una sau mai multe clase de bază se numește **clasa derivată**.
- I posibilitatea refolosirii lucrurilor care funcționează
- I organizează și facilitează polimorfismul și încapsularea
- I ” Anumite obiecte sunt similare dar în același timp diferite”.

# Principii POO

## Moștenirea

- | Proprietatea de **moștenire**: proprietatea claselor prin care o clasă nou construită poate prelua datele și metodele clasei mai vechi.
- | Clasa **derivată** se află întotdeauna pe un nivel imediat inferior celui corespunzător clasei de **bază**.
- | **În Java există doar moștenire simplă**, o ierarhie de clase în care fiecare clasă derivată are o singură clasă de bază.

Exemplu.



- | Clasa *dreptunghi* este o clasă derivată (**subclasă**) a clasei *paralelogram*, iar clasa *paralelogram* este o clasă de bază (**supraclasă**) a clasei *dreptunghi*.
- | Astfel, o ierarhie de concepte conduce la o **ierarhie între clasele** care implementează conceptele ierarhice respective.

## Principii POO: Polimorfismul, supraîncărcarea

- I Mai multe funcții pot avea același nume în același domeniu de definiție, dacă se pot diferenția prin numărul sau tipul argumentelor de apel.
- I O funcție este **polimorfică** dacă se poate executa cu același efect asupra unor valori de tipuri diferite (ex. operatorul & din C)
- I Un alt mecanism este **supraîncărcarea funcțiilor** (function overloading).
- I O funcție este supraîncărcată dacă execută operații diferite în contexte diferite (ex. operatorul '+' din Java)
- I Se poate aplica doar funcțiilor.
- I **Supradefinirea** (overriding) oferă posibilitatea de a redefini metode pentru clasele derivate, metodele au același tip și aceeași parametri.

## Principii POO: Polimorfismul, supraîncărcarea

- I Dacă în același domeniu sunt definite mai multe funcții cu același nume, la fiecare apel se selectează funcția corectă prin compararea tipurilor argumentelor reale de apel cu tipurile argumentelor formale ale funcției.
  - I `double abs(double);`
  - I `int abs(int);`
  - I `abs(1); // apeleaza abs(int)`
  - I `abs(1.0); // apeleaza abs(double)`
- I Nu este admis ca funcțiile să difere doar prin tipul returnat.
- I Două funcții declarate cu același nume se referă la aceeași funcție dacă sunt în același domeniu și au număr și tipuri identice de argumente.

# Concluzii POO

---

- I Ideea POO este de a crea programele ca o colecție de obiecte, unități individuale de cod care interacționează unele cu altele, în loc de simple liste de instrucțiuni sau de apeluri de proceduri.
- I Obiectele POO sînt de obicei reprezentări ale obiectelor din viața reală, astfel încît programele realizate prin tehnica POO sînt mai ușor de înțeles, de depanat și de extins decît programele procedurale (mai ales în cazul proiectelor software complexe și de dimensiuni mari, care se gestionează făcînd apel la ingineria programării).



# Tip abstract de date

---

- mulțime de date care au aceeași reprezentare și pentru care este definit setul de operații care se pot executa asupra elementelor mulțimii respective.
- are două părți:
  - o parte care definește reprezentarea datelor
  - o parte care definește operațiile asupra datelor respective.

# Noțiunea de clasă

- | O **clasă** definește un tip abstract de date.

Definiție clasă:

```
class nume{  
    lista_elementelor_membru  
}
```

Lista elementelor membru poate conține:

- § declarații de date;
- § implementari de funcții;
- § prototipuri de funcții abstracte.

Ø Datele declarate printr-o definiție de clasă se numesc **date membru**

Ø Funcțiile definite sau pentru care este prezent numai prototipul, în definiția clasei, se numesc **funcții membru** sau **metode**.

Ø Atât datele cât și metodele pot avea modificatori de acces

# Modificatorii de acces

- I **Modificatorii de acces** sunt cuvinte rezervate ce **controlează accesul** celorlalte clase la membrii unei clase. Specificatorii de acces pentru variabilele și metodele unei clase sunt: public, protected, private și cel implicit (la nivel de pachet).

Specificator	Clasa	Subcls	Pachet	Oriunde
<b>Private</b>	<b>X</b>			
<b>Protected</b>	<b>X</b>	<b>X*</b>		
<b>Public</b>	<b>X</b>	<b>X</b>	<b>X</b>	<b>X</b>
<b>Implicit</b>	<b>X</b>		<b>X</b>	

# Clasă

Exemplu:

```
class Complex {  
    // date membru  
    float real;  
    float imag;  
    // functii membru publice  
    public void atribuire(float x, float y) {  
        real = x; imag=y;  
    }  
    public double retreal() {  
        return real;  
    }  
    public void afiscomplex(){  
        System.out.println(real+" "+"+imag+"*i");  
    }  
}
```

# Obiecte

---

- I Un *obiect* este o dată de un tip definit printr-o clasă. Se spune că obiectul este o *instanțiere* a clasei respective.
- I Formatul declarației unui obiect:  
*nume\_clasă nume\_obiect;*
- I Instanțierea obiectelor se face folosind operatorul *new*.  
*nume\_obiect = new nume\_clasă(..);*

# Obiecte

---

- ∅ **Datele membru** se alocă distinct la fiecare instanțiere a clasei. O excepție o constituie datele membru care au clasa de memorare *static*, ea este o parte comună pentru toate instanțierile clasei și există într-un singur exemplar.
- ∅ **Funcțiile membru** sunt într-un singur exemplar oricâte instanțieri ar exista. Legătura dintre funcții membru și obiectul pentru care se face apelul se realizează folosind operatorul **punct**.

# Obiecte

- I Exemplu de instanțieri pentru clasa *complex*:

```
Complex z;  
z=new Complex();
```

- I Atunci:

```
z.atribuire(0,0);  
z.afiscomplex();
```

- afișează numărul complex  $z$  (în cazul de față  $0+0i$ ).

# Constructori

- § Obiectele se generează și se pot inițializa la instanțiere cu ajutorul *constructorilor*
  - Ø Funcții membru de același nume cu numele clasei
  - Ø Funcții apelate automat la crearea obiectelor.
- § Valorile de inițializare se transferă constructorului și ele joacă același rol ca parametrii efectivi de la apelurile funcțiilor obișnuite.
- § Se pot defini mai mulți constructori pentru o clasă. În acest caz ei au același nume, dar diferă prin numărul și/sau tipurile parametrilor.



# Constructori

---

- § Dacă există mai mulți constructori, atunci la inițializare se utilizează regulile de la apelurile funcțiilor supraîncărcate.
- § Funcțiile constructor nu întorc valori, dar nu sunt precedați de cuvântul *void*.
- § Dacă clasa nu conține constructori, se generează un constructor fără parametri, adică un *constructor implicit*. El are rolul numai de alocare a obiectelor clasei respective, fără a le inițializa.

# Constructori

Exemplu:

```
class Complex {  
    double real;  
    double imag;  
    public Complex(double x, double y)  
        {real = x; imag = y;}  
    public Complex ( )  
        {real = 0; imag = 0;}  
}
```

Exemple de instanțiere:

```
Complex z= new Complex();           // z = 0 + 0*i  
Complex z1= new Complex(1,0);     // z1 = 1 + 0*
```

## Realizarea încapsulării datelor

- I accesul la datele sau funcțiile membre ale unei clase din orice punct al domeniului de definiție al clasei s-ar putea rezolva simplu prin declararea de tip public a acestora.
- I o astfel de implementare nu respectă principiul încapsulării datelor și se recomandă să fie evitată .
- I Din punct de vedere al dreptului de acces la membrii clasei, o clasă bine definită permite încapsularea (sau ascunderea informațiilor), prin care un obiect poate ascunde celor care-l folosesc modul de implementare, prin interzicerea accesului la datele și funcțiile private sau protected.

## Realizarea încapsulării datelor

- I În general, respectând principiul încapsulării, **datele membre sunt declarate private sau protected și nu pot fi accesate direct (pentru citire sau scriere) din funcții nemembre ale clasei.**
- I Pentru citirea sau modificarea unora dintre datele membre protejate în clasa respectivă se pot prevedea funcții membre de tip public, care pot fi apelate din orice punct al domeniului de definiție al clasei și fac parte din interfața clasei.
- I De exemplu, pentru clasa Complex , o implementare care respectă principiul încapsulării, dar, în același timp permite accesul la datele private ale clasei poate arăta astfel:

# Realizarea încapsulării datelor

```
class Complex {  
    private double real;  
    private double imag;  
    public Complex(double x, double y)  
        {real = x; imag = y;}  
    public Complex ( )  
        {real = 0; imag = 0;}  
    public void set(double x, double y)  
        { re = x; im = y; }  
    public void setre(double x)  
        { re = x;}  
    public void setim(double y)  
        { im = y; }  
    public double getre(){  
        return re;}  
    public double getim() {  
        return im;}  
}
```

# Realizarea încapsulării datelor

```
public void display(){
    System.out.println(re+" "+im+"i");
}

class test{
    public static void main(String arg[]){
        Complex c1=new Complex(),c2=new
        Complex(1,1);
        c1.set(7.2, 9.3);
        c1.display(); // afiseaza 7.2 9.3
        c1.setre(1.3);
        c1.setim(2.8);
        c1.display(); // afiseaza 1.3 2.8 }
    }
}
```

# Clase abstracte

- I clasele pentru care programatorul nu intenționează să instanțeze obiecte.
- I clase de bază in ierarhii
- I se mai numesc *clase abstracte de bază*.
- I o clasă devine abstractă dacă conține una sau mai multe declarații de funcții abstracte.
- I dacă o clasă ce moștenește o clasă ce conține o funcție abstractă nu implementează acea funcție abstractă moștenită, devine ea însăși o *clasă abstractă*.

OBS: clasele pentru care se pot instanția obiecte se numesc *clase concrete*.