OC. Elemente de programare orientata spre obiecte

OC.1. Conceptul de obiect

In lumea in care traim suntem obisnuiti sa numim **obiecte** acele *entitati care sunt caracterizate prin masa, adica materie*. **Prin extensie**, pot fi definite *alte obiecte fara masa, care sunt mai degraba concepte* decat entitati fizice. **Tot prin extensie**, *obiectele pot apartine unei lumi virtuale*.

Obiectul este unitatea atomica formata din *reuniunea unei stari si a unui comportament*. El prezinta o relatie de **incapsulare** care asigura o *coeziune interna foarte puternica* si un *cuplaj slab cu exteriorul*.
Obiectul isi indeplineste rolul si responsabilitatea sa cu adevarat doar in momentul in care, *prin intermediul trimiterii unor mesaje*, el *participa la un scenariu de comunicatie*.

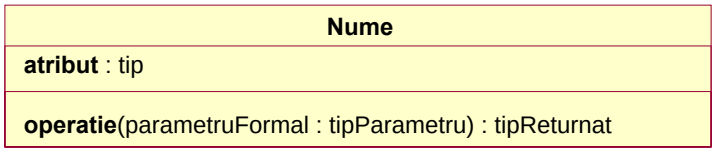
Obiectele informatice definesc o *reprezentare abstracta a entitatilor unor lumi reale sau virtuale*, cu *scopul de a conduce sau simula*. Aceasta reprezentare abstracta reda o *imagine simplificata a unui obiect care exista in lumea perceputa de utilizator*. Obiectele informatice, pe care le numim in general obiecte, **incapsuleaza** o *parte din cunoasterea lumii in care ele evolueaza*.

Utilizatorii tehnologiilor OO obisnuiesc sa considere obiectele ca fiind animate de o "viata" proprie, astfel incat ele li se arata adesea intr-o perspectiva antropomorfa. Fiind "vii", obiectele lumii reale se nasc, traiesc si mor. **Modelarea OO** permite *reprezentarea ciclului de viata al obiectelor de-a lungul interactiunii lor*.

OC.2. Conceptele de bază ale abordării OO

In cele ce urmeaza vom utiliza anumite notatii definite in limbajul UML (Unified Modeling Language), limbaj ce va fi prezentat in capitolele urmatoare, pentru a ilustra anumite concepte. Deoarece anumite concepte vor fi illustrate si in limbajul Java, vor fi facute si anumite paralele intre simbolurile UML si constructiile sintactice Java.

Clasa este **o colectie de elemente de date numite atribute** (variabile membru, proprietati, campuri, etc.) **si de operatii** (functii membru, metode, etc.). **Simbolul UML al unei clase** poate contine maximum 3 casete dreptunghiulare suprapuse, dintre care cea de sus contine numele clasei, cea din mijloc atributele, iar cea de jos operatiile:



Simbolului UML anterior ii va corespunde **codul Java**:

```
1    class Nume {
2       tip atribut;
3
4       tipReturnat operatie(tipParametru parametruFormal) {
5         // corpul operatiei – returneaza valoare de tipul tipReturnat
6       }
7    }
```

Formatul declaratiei (semnaturii) unei operatii UML (cu un singur argument) este (ca in limbajul **Pascal**):

```
numeOperatieUML(numeArgumentUML : tipArgumentUML) : tipReturnatUML
```

pe cand **formatul declaratiei unei metode Java** (cu un singur argument) este (ca in **C** si **C++**):

```
tipReturnatJava numeMetodaJava(tipArgumentJava numeArgumentJava)
```

Formatul declaratiei unui atribut UML este (ca in limbajul **Pascal**):

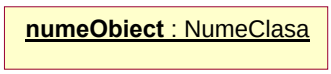
```
numeAtributUML : tipAtributUML
```

pe cand **formatul declaratiei unui atribut Java** este (ca in limbajele **C** si **C++**):

```
tipAtributJava numeAtributJava
```

Clasa reprezinta **tipul (domeniul de definitie) pentru variabile numite obiecte**.

Un **obiect** este reprezentat **in UML**, ca un dreptunghi in care este plasat numele obiectului subliniat, urmat de simbolul ":", si de numele clasei careia ii apartine, de asemenea subliniat:



ceea ce **in Java** are **ca echivalent declaratia unei variabile obiect** numita **`numeObiect`** de tipul **`NumeClasa`**:

```
NumeClasa numeObiect;
```

In diagrama UML, sub numele obiectului pot fi plasate perechi atribut-valoare sub forma:

```
numeAtribut = valoare
```

OC.2.1. Obiectele

Obiectul este **încapsularea** unei **stări** şi a unui **comportament**. Obiectul reprezinta mai mult insa decat simpla insumare a acestora. Pot exista si obiecte fara stare sau fara comportament, insa toate obiectele au o identitate. Putem asadar spune ca **obiectul** este o ***reprezentare abstractă*** a unor entităţi reale sau virtuale, caracterizată de:

- ***identitate***, prin care e deosebit de alte obiecte, implementata in general ca **variabila obiect**,
- ***comportament vizibil***, accesibil altor obiecte, implementata in general ca **set de functii** (membru),
- ***stare internă ascunsă***, proprie obiectului, implementata in general ca **set de variabile** (membru).

OC.2.1.1. Starea obiectului

Starea obiectului regrupeaza **valorile instantanee ale tuturor atributelor unui obiect**.

Atributul fiind o informatie care **califica (spune ceva despre) obiectul caruia ii apartine**.

Informatia este stocata intr-o variabila numita si **variabila membru** (sau **proprietate**, **camp**, etc.) a obiectului.
Fiecare atribut poate lua valori intr-un domeniu definitie dat (**tipul variabilei membru**).

Starea unui obiect, la un moment dat, corespunde unei **selectii de valori posibile ale diferitelor atribute**.

Pentru ilustrare, sa consideram **exemplul unui obiect care incapsuleaza informatiile privind pozitia unui punct intr-un plan** (obiect care poate fi folosit, de exemplu, pentru a pastra informatiile privind un element de imagine).

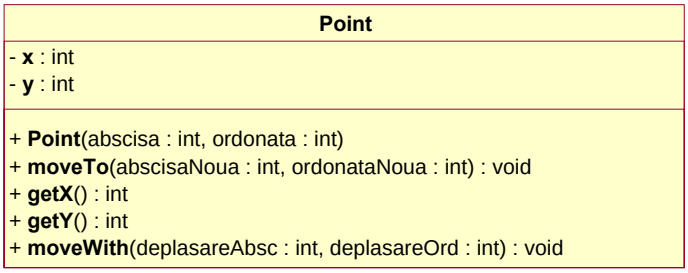
Mai jos este **definitia Java a tipului (clasei) unui astfel de obiect**.

```
1    public class Point {
2      // atribute (variabila membru)
3      private int x;
4      private int y;
5      // operatie care initializeaza atributele = constructor Java
6      public Point(int abscisa, int ordonata) {
7        x = abscisa;
8        y = ordonata;
9      }
10     // operatie care modifica atributele = metoda (functie membru) Java
11     public void moveTo(int abscisaNoua, int ordonataNoua) {
12       x = abscisaNoua;
13       y = ordonataNoua;
14     }
15     // operatie care modifica atributele = metoda (functie membru) Java
16     public void moveWith(int deplasareAbsc, int deplasareOrd) {
17       x = x + deplasareAbsc;
18       y = y + deplasareOrd;
19     }
20     // operatii prin care se obtin valorile atributelor = metode Java
21     public int getX() {
22       return x;
23     }
24     public int getY() {
25       return y;
26     }
27   }
```

| Declaratii (specificare) variabile

| Semnaturi (declaratii, specificari) operatii + Implementari (corpuri) operatii

Simbolul UML corespunzator definitiei de clasa **Java** de mai sus este:

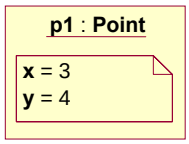


Declaratia Java:

```
Point p1 = new Point(3, 4);
```

va conduce la crearea unui obiect **p1** de tip **Point** ale carui atribute au valorile **x=3** si respectiv **y=4**.

Echivalentul UML:



Astfel, **obiectul p1 de tip Point incapsuleaza informatiile privind un punct in plan de coordonate {3, 4}**. **Starea obiectului p1** este asadar perechea de coordonate **{3, 4}**.

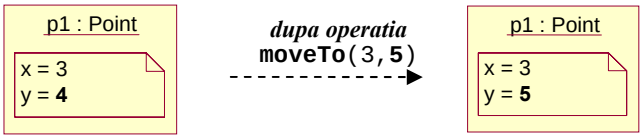
Starea unui obiect evolueaza in decursul timpului. Totusi anumite componente ale starii pot ramane nemodificate. Starea obiectului este variabila si poate fi vazuta ca o **consecinta a comportamentului** sau **trecut**.

Secventa Java:

```
Point p1 = new Point(3, 4);
p1.moveTo(3, 5);
```

va schimba starea obiectului `p1` in perechea de coordonate {3, 5}, ceea ce este echivalent cu deplasarea ordonatei punctului (departarea cu 1 a punctului de abscisa).

Echivalentul UML:



OC.2.1.2. Comportamentul obiectului

Comportamentul regrupeaza toate **posibilitatile de evolutie a unui obiect** adica **actiunile si reactiile acestui obiect**.

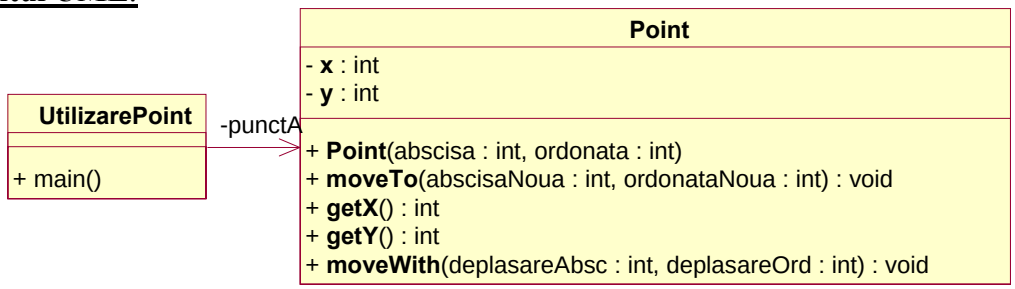
Un atom de comportament este denumit **operatie**. Operatiile sunt implementate ca **functii membru** ale obiectului sau **metode**.

Operatiile unui obiect **sunt declansate ca urmare a unei stimulari externe**, reprezentate sub forma unui **mesaj** care este **trimis de catre un alt obiect** (care ii apeleaza/invoca functiile membru/metodele).

Exemplu de comportament (utilizare obiect), in Java. Sa luam **exemplul clasei Java** `UtilizarePoint` **care exemplifica operatii asupra obiectelor clasei** `Point`:

```
1    public class UtilizarePoint {
2      private static Point punctA;                        // atribut de tip Point
3
4      public static void main(String[] args) {            // declaratie metoda
5                                                          // corp metoda
6        punctA = new Point(3, 4); // alocare si initializare atribut punctA
7
8        punctA.moveTo(3, 5);      // trimitere mesaj moveTo() catre punctA
9
10       punctA.moveWith(3, 5);    // trimitere mesaj moveWith() catre punctA
11     }
12   }
```

Echivalentul UML:



Interactiunile intre obiecte pot fi reprezentate prin intermediul unor diagrame (numite diagrame de colaborare intre obiecte) in care obiectele care interactioneaza sunt legate intre ele prin linii continue denumite **legaturi**.

In **diagrama UML** urmatoare, in functie de valoarea mesajului, se declanseaza fie operatia `moveTo()` fie operatia `moveWith()`.

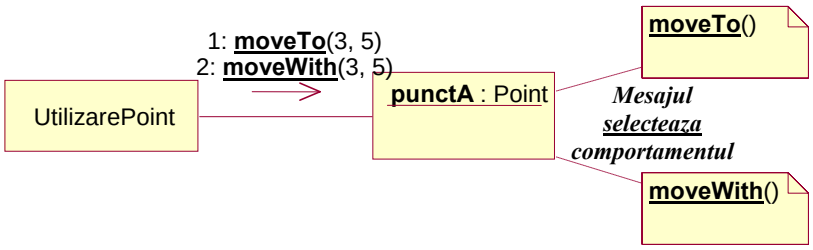


Fig. x. Mesajul serveste ca selector al operatiei de executat

Prezenta unei legaturi (linii) semnifica faptul ca **un obiect cunoaste sau vede un alt obiect**, ca ii poate apela/invoca functiile membru/metodele, ca **poate comunica cu acesta prin intermediul mesajelor declansate** de apelurile/invocarile functiilor membru/metodelor.

Mesajele navigheaza de-a lungul legaturilor, implicit in ambele directii.

Starea si comportamentul sunt **dependente**. **Comportamentul** la un moment dat **depinde de starea curenta**. **Starea** poate fi **modificata prin comportament**.

Cat timp un avion este "La sol", nu e posibil a-l face sa aterizeze. Altfel spus, **in starea "La sol" comportamentul Aterizare nu e valid.**

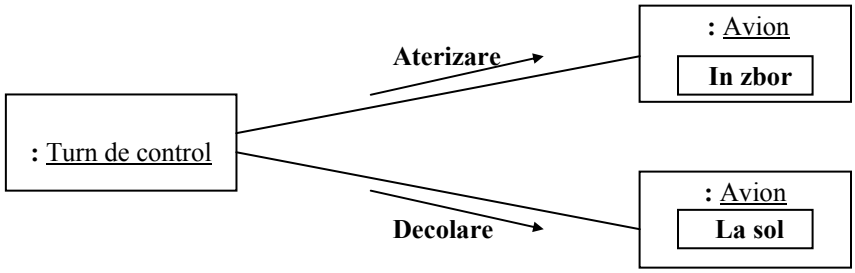


Fig. x. Dependenta comportamentului (realizarii operatiilor) de stare (valorilor atributelor).

In starea "In zbor" comportamentul Aterizare e posibil. El **conduce la schimbarea starii**, din "In zbor" in "La sol".
Dupa aterizare, **starea fiind "La sol", operatia Aterizare nu mai are sens**.

OC.2.1.3. Identitatea obiectului

Existenta proprie a unui obiect este caracterizata de **identitate**, care permite **distingerea tuturor obiectelor** intr-o maniera non-ambigua si independenta de starea lor. Astfel pot fi **tratate distinct** doua obiecte ale caror atribute au valori identice.

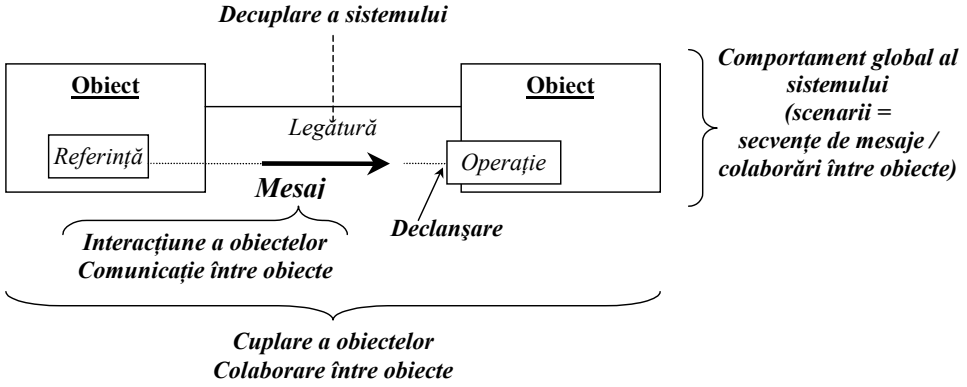
In faza de **modelare** identitatea nu se reprezinta intr-o forma specifica, obiectul avand o **identitate implicita**. In faza de **implementare**, identitatea este adesea construita utilizand un **identificator** (**variabila obiect**) rezultat natural din domeniul problemei.

OC.2.2. Mesajele si comunicația între obiecte

Sistemele informatice OO pot fi vazute ca **societăți de obiecte** care **conlucrează** (**colaborează**) pentru a realiza funcțiile aplicației.

Conlucrarea se bazează pe **comunicația între obiectele componente**.

Mesajul este forma de reprezentare a **stimulului extern care declanşează o operație**.



***Mesajul** este **unitatea de comunicație** între obiecte, **suportul unei relații de comunicație** care **leagă în mod dinamic obiecte** care au fost **separate prin procesul de descompunere**.*

Legătura este o **cale între obiectele care se cunosc** (văd) unul pe altul (îşi pot transmite mesaje), pentru aceasta avand referințe unul către celălalt.

Legătura statică este realizată **la compilare (*compile-time*)**, **legatura dinamică** este creată **în timpul execuției (*run-time*)**.

Interacțiunea este **rezultatul (posibilității) transmiterii mesajelor între obiecte legate**.

Mesajul permite **interacțiunea flexibilă**, fiind simultan **agent de cuplaj** şi **agent de decuplare**. Mesajul este un **integrator dinamic** care compune o funcție a aplicației prin **punerea în colaborare a unui grup de obiecte**.

OC.2.3. Clasa (tipul de date al obiectelor)

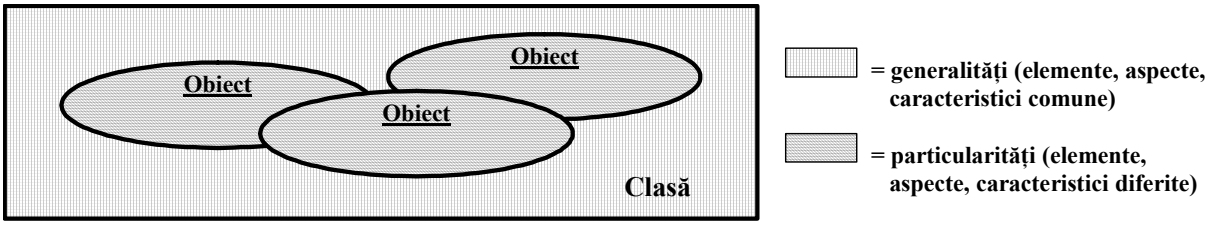
OC.2.3.1. Abstractizarea

Abstractizarea este **capacitatea** (aptitudinea, caracteristica) **umană** care constă în **concentrarea gândirii pe un element sau aspect** al unei reprezentări sau al unei noțiuni, **atenția îndreptându-se în special asupra acestuia şi neglijându-le pe toate celelalte**.

Abstractizarea OO înseamnă **identificarea caracteristicilor comune** ale unui ansamblu de elemente (**obiecte**), urmată de **descrierea într-o formă condensată a acestor caracteristici** în ceea ce se numeşte **clasă**.

Abstractizarea este **arbitrară**, deoarece ea **se defineşte în raport cu un punct de vedere**.

Generalitățile (elementele comune obtinute prin abstractizare**)** sunt descrise **în clasă** iar **particularitățile (elementele distincte)** sunt descrise **în obiecte**.



Sa analizam, **pentru exemplificare, o parte din codul** clasei **`DatagramPacket`** (am compactat continuturile anumitor metode, pentru simplificarea intelegerii codului):

```
1    // Cod care face parte din pachetul claselor Java pentru comunicatii (retea)
2
3    package java.net;
4
5    // Clasa care incapsuleaza pachete UDP (datagrame).
6    // Clasa finala, nu poate fi extinsa prin mostenire
7
8    public final class DatagramPacket {
9
10     // Atribute, accesibile tuturor claselor in pachetul java.net
11
12     byte[] buf;           // tabloul de octeti care contine datele pachetului
13     int offset;           // indexul de la care incep datele pachetului
14     int length;           // numarul de octeti de date din tabloul de octeti
15     int bufLength;        // lungimea tabloului de octeti
16     InetAddress address;  // adresa IP a masinii sursa/destinatie a pachetului
17     int port;             // portul UDP al masinii sursa/destinatie a pachetului
18
19     // Constructori – initializeaza obiectele de tip DatagramPacket
20
21     // Initializeaza un DatagramPacket pentru pachete de receptionat,
22     public DatagramPacket(byte buf[], int length) {
23       this.buf = buf;
24       this.length = length;
25       this.bufLength = length;
26       this.offset = 0;
27       this.address = null;
28       this.port = -1;
29     }
30
31     // Initializeaza un DatagramPacket pentru pachete de trimis
32     public DatagramPacket(byte buf[], int length, InetAddress address, int port) {
33       this.buf = buf;
34       this.length = length;
35       this.bufLength = length;
36       this.offset = 0;
37       this.address = address;
38       this.port = port;
39     }
40
41     // Alti constructori, alte metode...
42
43     // Metoda - Returneaza adresa IP a masinii sursa/destinatie a acestui pachet
44     public synchronized InetAddress getAddress() {
45         return this.address;
46     }
47
48     // Metoda - Returneaza portul UDP al masinii sursa/destinatie a acestui pachet
49     public synchronized int getPort() {
50         return this.port;
51     }
52 }
```

Dupa cum se poate observa, **orice obiect din clasa** (de tipul) **`DatagramPacket`** are **6 atribute**:
- **`buf`** – un tablou de octeti in care sunt plasate datele care formeaza pachetul,
- **`offset`** – indexul in tabloul `buf` de la care sunt plasate datele,
- **`length`** – numarul de octeti de date din tabloul `buf`,
- **`bufLength`** – lungimea tabloului `buf` (numarul de octeti de date care pot fi plasati in `buf`),
- **`address`** – adresa IP (incapsulata intr-un obiect de tip `InetAddress`) a masinii destinatie,
- **`port`** – portul UDP al masinii catre care se trimite pachetul (destinatie).

Acele obiecte de tip **`DatagramPacket`** care sunt **folosite pentru trimiterea pachetelor au nevoie la initializare de specificarea** adresei IP si portului UDP ale **masinii destinatie a pachetului**.

De exemplu:

```
byte[] bufferDate = new byte[1024];
DatagramPacket packetT = new DatagramPacket(bufferDate, bufferDate.length,
                             InetAddress.getByName("nume.elcom.pub.ro"), 2000);
```

obiectul **`packetT`** este initializat pentru a putea trimite cel mult 1024 octeti plasati in tabloul de octeti **`bufferDate`** catre portul UDP **`2000`** al masinii **`nume.elcom.pub.ro`**.

Obiectele de tip **`DatagramPacket`** care sunt **folosite pentru receptia pachetelor nu au nevoie la initializare de specificarea** adresei IP si portului UDP ale **masinii sursa a pachetului**.

De exemplu:

```
byte[] bufferDate = new byte[1024];
DatagramPacket packetR = new DatagramPacket(bufferDate, bufferDate.length);
```

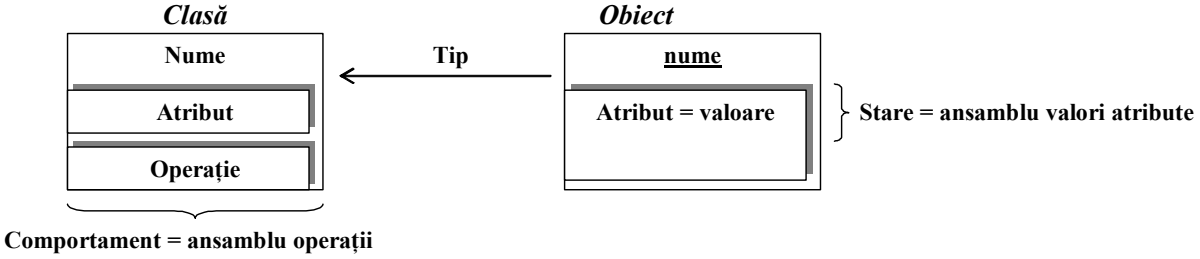
obiectul **`packetR`** este initializat pentru a putea primi cel mult 1024 octeti in tabloul de octeti **`bufferDate`** de la orice masina. **Dupa receptia pachetului UDP**, prin utilizarea metodelor **`getAddress()`** si **`getPort()`** pot fi obtinute **adresa IP si portul UDP ale masinii sursa** a pachetului.

Clasa `DatagramPacket` contine in definitia sa **elementele comune** (cele 6 atribute) **ale tuturor obiectelor** pachet UDP.

Obiectele pachet UDP create ca variabile avand ca tip clasa **`DatagramPacket`** **contin detalii** care **le particularizeaza**. Dupa cum tocmai am aratat, **felul in care sunt initializate atributele adresa IP si port UDP** ale obiectelor clasei **`DatagramPacket`**, **diferentiaza aceste obiecte** in pachete utilizabile **pentru receptie** si pachete utilizabile **pentru transmisie**.

OC.2.3.2. Relațiile dintre clasă şi obiecte

Clasa descrie **domeniul de definiție al unui ansamblu de obiecte**, fiind **tipul de date abstract** al ansamblului de obiecte.



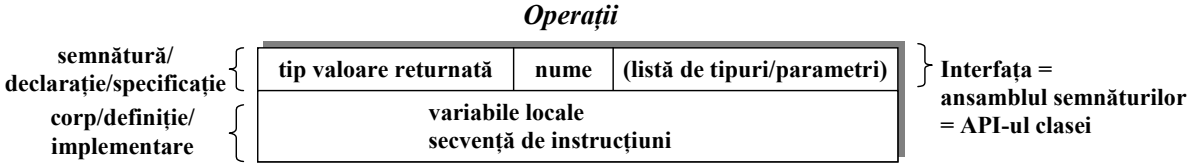
Construirea obiectelor informatice pornind de la clase poartă numele de **instanțiere** sau **exemplificare**, obiectele fiind **instanțe** (de la *instance*) sau **exemple** ale unor clase.

OC.2.3.3. Separarea interfeței de implementare în descrierea claselor

Descrierea claselor cuprinde:

- specificația (interfața) clasei, care:
- descrie **domeniul de definiție** și **proprietățile instanțelor** clasei,
- corespunde noțiunii de **tip**, așa cum este el definit în limbajele de programare clasice;

- implementarea (realizarea) clasei, care:
- descrie **modul în care este implementată specificarea**,
- descrie **conținutul corpului** fiecărei operații și **datele necesare** funcționării acesteia.



Clasa stabilește un **contract** cu alte clase, în care **se angajează să furnizeze serviciile publice care formează specificația sa (API** = ***Application Programming Interface***), celelalte clase angajându-se să nu utilizeze alte cunoștințe decât cele descrise în această specificație.

Separarea între specificație și implementare participă la **ridicarea nivelului de abstracție al clasei**:

- **informațiile importante** sunt descrise în **specificație (vizibilă, accesibilă, publică)**, și
- **detaliile** sunt precizate în **implementare (ascunsă, inaccesibilă, privată)**.

Exemplul numerelor complexe ilustrează bine separarea specificatiei de implementare. **Specificatia generala a unei clase `Complex`**:

```
 1    public class Complex {
 2
 3      // Atribute private (ascunse, inaccesibile din exteriorul clasei)
 4
 5      // Constructor – initializeaza obiectele de tip Complex
 6      public Complex(float real, float imag) {
 7        // Implementare
 8      }
 9      // Constructor – initializeaza obiectele de tip Complex
10      public Complex(double modul, double faza) {
11        // Implementare
12      }
13      // Returneaza partea reala
14      public double getReal() {
15        // Implementare
16      }
17      // Returneaza partea imaginara
18      public double getIumag() {
19        // Implementare
20      }
21      // Returneaza modulul
22      public double getModul() {
23        // Implementare
24      }
25      // Returneaza faza
26      public double getFaza() {
27        // Implementare
28      }
29    }
```

Implementarea carteziana (**atributele ascunse** sunt **coordonatele carteziene**) a clasei **`Complex`**:

```
1    public class Complex {
2
3      // Atribute private (ascunse, inaccesibile din exteriorul clasei)
4      private double real;           // partea reala (abscisa)
5      private double imag;           // partea imaginara (ordonata)
6
7      public Complex(float real, float imag) {
8        this.real = real;
9        this.imag = imag;
10     }
11     public Complex(double modul, double faza) {
12       this.real = modul * Math.cos(faza);
13       this.imag = modul * Math.sin(faza);
14     }
15     public double getReal() {
16         return this.real;
17     }
18     public double getImag() {
19         return this.imag;
20     }
21     public double getModul() {
22         return Math.sqrt(this.real*this.real + this.imag*this.imag);
23     }
24     public double getFaza() {
25         return Math.atan2(this.real, this.imag);
26     }
27   }
```

Implementarea polara (**atributele ascunse** sunt **coordonatele polare**) a clasei **`Complex`**:

```
1    public class Complex {
2
3      // Atribute private (ascunse, inaccesibile din exteriorul clasei)
4      private double modul;              // modulul (raza)
5      private double faza;               // faza (unghiul)
6
7      public Complex(float real, float imag) {
8        this.modul = Math.sqrt(real*real + imag*imag);
9        this.faza = Math.atan2(real, imag);
10     }
11     public Complex(double modul, double faza) {
12       this.modul = modul;
13       this.faza = modul;
14     }
15     public double getReal() {
16         return this.modul*Math.cos(this.faza);
17     }
18     public double getImag() {
19         return this.modul*Math.sin(this.faza);
20     }
21     public double getModul() {
22         return this.modul;
23     }
24     public double getFaza() {
25         return this.faza;
26     }
27   }
```

In cazul de mai sus:

- **specificația nu este afectată de schimbarea reprezentării interne** (polară sau carteziană),
- **obiectele utilizator** al obiectelor instanțe ale clasei numerelor complexe **cunosc doar specificația şi nu sunt nici ele afectate de schimbarea reprezentării interne**.

OC.2.3.4. Încapsularea (regruparea elementelor, protectia si ascunderea detaliilor)

Incapsularea inseamna, pe langa regruparea unor elemente (de date – atributele - si de comportament - operatiile) si **ascunderea si protectia detaliilor**.

Avantajele încapsulării:

- **datele încapsulate** în obiecte sunt **protejate** de accesul nedorit, fiindu-le **garantată integritatea**,
- **utilizatorii** unei abstracții **nu depind de implementarea** acestei abstracții **ci de specificația** ei, ceea ce **reduce cuplajul** între modele.

Exemplul numerelor complexe ilustrează si ascunderea detaliilor.

Implicit:

- **valorile atributelor unui obiect sunt ascunse** în obiecte şi **nu pot fi manevrate direct** de către alte obiecte (**atributele sunt implicit private**),
- **operatiile sunt publice,** toate **interacțiunile între obiecte** sunt efectuate **declanşând diversele operații declarate în specificația clasei** şi accesibile altor obiecte.

Urmatorul **cod Java**:

```
Complex c1 = new Complex(2, -2);
System.out.println("Coordonatele carteziene ale lui c1 sunt {" +
                                c1.getReal() + ", " + c1.getImag() + "}");
System.out.println("Coordonatele polare ale lui c1 sunt {" +
                                c1.getModul() + ", " + c1.getFaza() + "}");
```

va conduce la acelasi rezultat, indiferent de implementarea clasei **`Complex`**.

Utilizatorii clasei **`Complex`** nu pot afla cum arata **implementarea interna a clasei**, care este **ascunsa**.

Metodele `getReal()`, `getImag()`, `getModul()` si respectiv `getFaza()` **permit accesul la informatia incapsulata**, pe cand **atributele sunt inaccesibile in mod direct**, iar **detaliile privind implementarea fiind complet inaccesibile**.

OC.2.4. Relații între clase

OC.2.4.1. Relații si asocieri între clase

Legăturile particulare care unesc obiectele pot fi văzute la modul abstract în lumea claselor, **fiecărei familii de legături între obiecte ale aceloraşi clase** corespunzându-i **o relație între clasele acelor obiecte**.

Legăturile sunt **instanțe ale relațiilor** între clase.

Asocierea este **abstracția legăturilor** care există între obiectele instanțe ale claselor asociate, este implicit **bidirecțională, nu este conținută** şi **nici subordonată claselor**.

Relația exprimă o **formă de cuplaj între abstracții**, **forța acestui cuplaj depinzând de natura relației** în doemniul problemei.

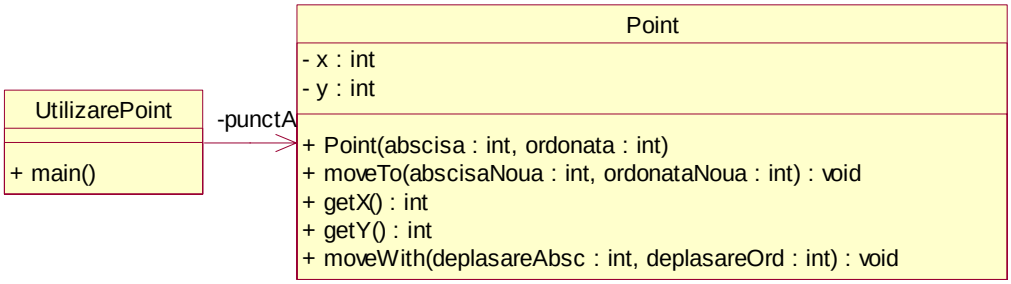
Implicit, asocierea exprimă un cuplaj redus, clasele asociate rămânând relativ independente una de alta.

Clasele `Point` si **`UtilizarePoint`** sunt de exemplu intr-o **relatie de asociere (cu navigabilitate) unidirectionala**, clasa **`UtilizarePoint`** avand un atribut **`pointA`** de tip **`Point`** care permite clasei **`UtilizarePoint`** sa trimita mesaje unui obiect (**`pointA`**) al clasei **`Point`**.

```
private Point punctA;                                   // atribut de tip Point
```

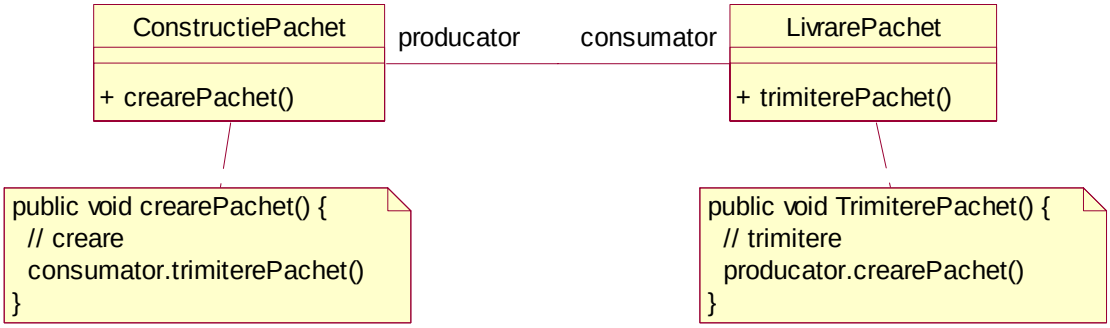
Clasa **`Point`** in schimb nu are nici o referinta catre clasa **`UtilizarePoint`** care sa ii permita trimiterea de mesaje (invocari de metode). **Legatura intre obiectele** celor doua clase **fiind unidirectionala, asocierea** dintre ele **este tot unidirectionala**.

In UML asocierile unidirectionale se reprezinta prin **sageti** indreptate pe directia pe care se pot trimite mesaje (catre care exista referinta).



Asocierile bidirectionale intre doua clase corespund situatiei in care **ambele clase au referinte una catre cealalta**.

Diagrama UML:



are drept corespondent **codul Java**:

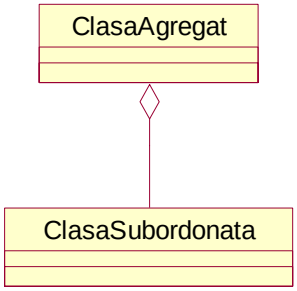
```
public class ConstructiePachet {
  LivrarePachet consumator;

  public void crearePachet(  /* eventuali parametri */ ) {
    // creare
    consumator.trimiterePachet()
  }
}

public class LivrarePachet {
  ConstructiePachet producator;

  public void trimiterePachet( /* eventuali parametri */ ) {
    // trimitere
    producator.crearePachet()
  }
}
```

OC.2.4.2. Agregarea claselor

Agregarea este o **formă particulară de asociere care exprimă un cuplaj strâns între clase**,

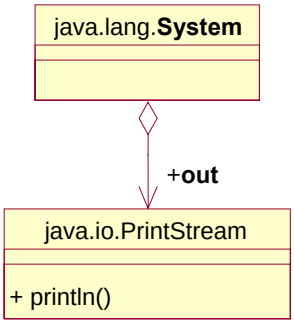
- **una dintre clase joacă un rol mai important** decât cealaltă,
- reprezintă **relaţii de subordonare** de tip "***master-slave***", "**întreg-părţi**" sau "**ansamblu-componentă**"

Notatia UML pentru agregare o **linie** care uneste simbolul claselor, **terminata cu un romb in capatul dinspre clasa cu rol mai important** (agregat).



Un bun exemplu este obiectul **`out`**, un **atribut cu caracter global** (`static`, de clasa) **al clasei *`java.lang.`*`System`**. Clasa **`System` incapsuleaza o parte din resursele** hardware si software ale **sistemului de executie.**

Obiectul **`out`**, al carui tip este clasa ***`java.io.`*`PrintStream`**, **incapsuleaza** informatiile privind **consola standard de iesire**, una dintre resursele sistemului de executie incapsulata in clasa **`System`.**

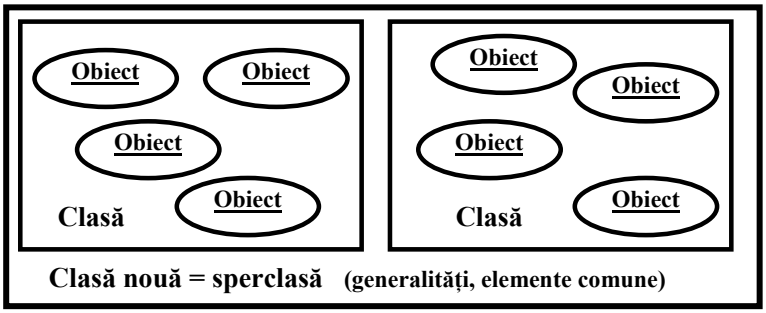


OC.2.5. Ierarhii de clase

OC.2.5.1. Generalizarea claselor

Generalizarea:

- **extragerea elementelor comune** (atribute, operaţii şi constrângeri) ale unui ansamblu de clase **într-o nouă clasă mai generală**, denumită **superclasă,**
- **superclasa** este **o abstracţie a subclaselor ei,**
- **arborii de clase** sunt **construiţi pornind de la frunze**
- **utilizată din momentul în care elementele modelului au fost identificate**, pentru a obţine o descriere detaşată a soluţiilor



Generalizarea semnifică "**este un**" sau "**este un fel de**", şi priveşte doar clasele, adică **nu este instanțiabilă**. De asemenea, generalizarea **nu este o relație reflexivă**, **este asimetrică şi tranzitivă**.

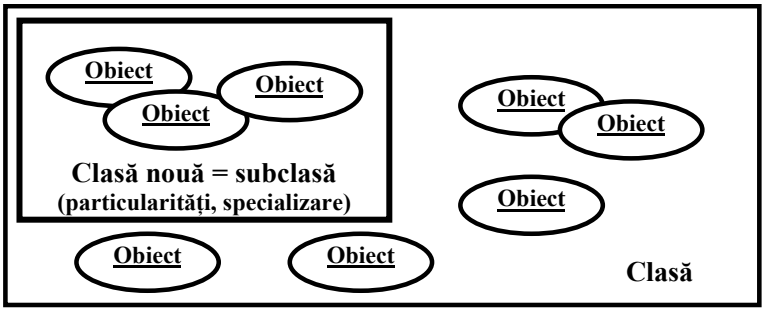
De fapt, generalizarea actioneaza in OO la doua niveluri:
- **clasele sunt generalizari ale ansamblurilor de obiecte** (un obiect este **de felul** specificat de o clasa),
- **superclasele sunt generalizari ale unor clase** (obiectele **de felul** specificat intr-o clasa sunt in acelasi timp si **de felul** specificat in superclasa).

Orientarea spre obiecte (OO) presupune **ambele tipuri de generalizare**, iar **limbajele orientate spre obiecte** sunt acelea care **ofera ambele mecanisme de generalizare**. Limbajele care ofera doar constructii numite obiecte (si eventual clase) se pot numi **limbaje care lucreaza cu obiecte** (si eventual clase).

OC.2.5.2. Specializarea claselor

Specializarea:

- **inseamna capturarea particularităților** unui ansamblu de obiecte nediscriminate ale unei clase existente, noile caracteristici fiind reprezentate **într-o nouă clasă mai specializată**, denumită **subclasă,**
- este utilă pentru **extinderea coerentă a unui ansamblu de clase**
- este **bază a programării prin extindere şi a reutilizării**, noile cerințe fiind încapsulate în subclase care extind în mod armonios (coerent) funcțiile existente



În **elaborarea unei ierarhii de clase**, se cer diferite **aptitudini sau competențe**:

- **capacitate de abstractizare**, independentă de cunoştințele tehnice, pentru **identificarea** superclaselor (pentru **generalizare**),
- **experiență şi cunoştințe aprofundate** într-un domeniu particular, pentru **implementarea** subclaselor (pentru **specializare**).

Pe de altă parte, există un **paradox**:

- e **dificil de găsit superclase**, dar programele scrise cu ajutorul lor sunt **mai uşor de dezvoltat**,
- e **destul de uşor de găsit subclase**, dar **dificil de implementat**.

OC.2.5.3. Clasificarea claselor

Ierarhiile de clase, sau **clasificările**, obtinute prin generalizare (sau specializare) servesc managementului complexității prin **ordonarea** obiectelor în **arborescențe de clase abstracte**.

Problema clasificării (J.J.Rousseau, 1755):

*"**Fiecare obiect primeşte mai întâi un nume particular, fără a se ţine seama de gen sau de tip**. Dacă un copac se numeşte A, un altul se numeşte B, căci **prima idee care vine privind două lucruri, este că ele nu sunt aceleaşi**, şi **este necesar adesea să trecă mult timp pentru a observa ceea ce au în comun**, astfel încât multe **cunoştinţe rămân limitate** şi multe **definiţii devin prea întinse**. Aglomerarea tuturor acestor denumiri **nu poate fi clasificată uşor**, căci pentru a le dispune sub denumiri comune şi generice, **este necesară cunoaşterea proprietăţilor şi a diferenţelor, sunt necesare observaţii şi definiţii**, adică de istorie şi metafizică, mult mai mult decât timpul de care dispune omul."*

Clasificarea trebuie să discrimineze obiectele, **să fie stabilă**, **să fie extensibilă**.

Clasificarea se realizează conform unor criterii care **depind de diverse perspective**, aşa încât **nu există clasificare, ci clasificări**, fiecare potrivită unei utilizări date.

Există numeroase **moduri de realizare a clasificării**, în programarea OO **tehnica cea mai utilizată** fiind **moştenirea între clase**.

OC.2.5.4. Moştenirea membrilor într-o ierarhie de clase

Moştenirea este o **tehnică de generalizare** oferită de limbajele de programare OO pentru a construi o clasă pornind de la una sau mai multe alte clase, **partajând atributele, operaţiile şi uneori constrângerile**, într-o ierarhie de clase.

In limbajul Java, orice clasa care nu extinde (prin mostenire) in mod explicit o alta clasa Java, **extinde (prin mostenire)** in mod **implicit clasa `Object`** (radacina ierarhiei de clase Java), clasa care contine metodele necesare tuturor obiectelor Java.

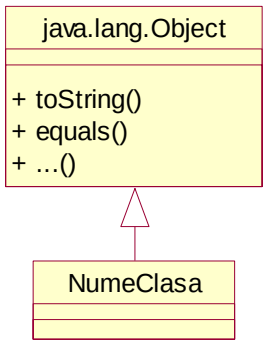
Urmatoarea **declaratie de clasa**:

```
class NumeClasa { // urmeaza corpul clasei ...
```

este **echivalenta cu**:

```
class NumeClasa extends Object { // urmeaza corpul clasei ...
```

Notatia UML pentru extinderea prin mostenire este o **linie care uneste clasa extinsa** (de baza, superclasa) **de clasa care extinde** (subclasa), linie **terminata cu un triunghi in capatul dinspre clasa de baza**. **Diagrama UML** corespunzatoare **codului Java** anterior:



Printre metodele declarate in clasa **`Object`** este si **`toString()`**, metoda care are ca **scop returnarea sub forma de `String` a informatiilor pe care le incapsuleaza obiectul caruia i se aplica aceasta metoda**.

In **cazul claselor de biblioteca Java**, metoda **`toString()`** returneaza ansamblul valorilor curente ale atributelor obiectului.

In **cazul claselor scrise de programator, in mod implicit** metoda **`toString()`** returneaza numele clasei careia ii apartine obiectul urmat de un cod alocat acelui obiect (*hashcode*). Implementarea implicita a metodei **`toString()`** este urmatoarea:

```
1    // Implementarea implicita a metodei toString(),
2    // mostenita de la clasa Object
3
4    public String toString() {
5       // (nu returneaza continutul ci numele clasei si codul obiectului!)
6       return getClass().getName() + "@" + Integer.toHexString(hashCode());
7    }
```

In **cazul in care programatorul doreste returnarea informatiilor incapsulate in obiect, trebuie specificat in mod explicit un nou cod (o noua implementare)** pentru metoda **`toString()`**. Acest lucru se obtine adaugand clasei din care face parte acel obiect o metoda cu declaratia:

```
public String toString() { // urmeaza corpul metodei ...
```

metoda care se spune ca **rescrie (*overrides*)** codul metodei cu acelasi nume din clasa extinsa (in acest caz clasa **`Object`**).

Dupa adaugarea acestei metode, apelul **`toString()` va conduce la executia noului cod**, pe cand apelul **`super.toString()` va conduce la executia codului din clasa extinsa** (**superclasa**, in acest caz codul implicit din clasa **`Object`**).

Printre metodele declarate in clasa **`Object`** este si metoda **`equals()`**, metoda care are ca scop **compararea obiectului caruia i se aplica aceasta metoda cu un obiect pasat ca parametru**, returnand valoarea booleana **`true`** in cazul egalitatii si valoarea booleana **`false`** in cazul inegalitatii celor doua obiecte.

In **cazul claselor de biblioteca Java**, metoda **`equals()`** compara ansamblul valorilor curente ale atributelor obiectului (continutul sau starea obiectului). Iata, de exemplu, implementarea metodei **`equals()`** in cazul clasei **`String`**.

```
1    // Implementarea explicita a metodei equals() in clasa String
2
3    public boolean equals(Object obj) {
4      // se verifica existenta unui parametru (obiect) non-null
5      // si faptul ca parametrul e obiect al clasei String
6      if ((obj != null) && (obj instanceof String)) {
7        String otherString = (String)obj;          // conversie de tip
8        int n = this.count;
9        if (n == otherString.count) {   // se compara numarul de caractere
10         char v1[] = this.value;
11         char v2[] = otherString.value;
12         int i = this.offset;
13         int j = otherString.offset;
14         while (n-- != 0)
15           if (v1[i++] != v2[j++]) return false; // se compara caracterele
16         return true;
17       }
18     }
19     return false;
20   }
```

In **cazul claselor scrise de programator, in mod implicit** metoda **`equals()`** compara referinta obiectului caruia i se aplica aceasta metoda cu referinta obiectului pasat ca parametru. Implementarea implicita a metodei **`equals()`** este urmatoarea:

```
1    // Implementarea implicita a metodei equals(),
2    // mostenita de la clasa Object
3
4    public boolean equals(Object obj) {
5      return (this == obj);     // (nu compara continutul ci referintele!!!)
6    }
```

In **cazul in care programatorul doreste compararea informatiilor incapsulate in obiect,** (ansamblul valorilor curente ale atributelor obiectului) **trebuie specificat in mod explicit un nou cod (o noua implementare)** pentru metoda **`equals()`**. Acest lucru se obtine adaugand clasei din care face parte acel obiect o metoda cu declaratia:

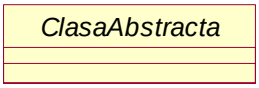
```
public boolean equals(Object obj) { // urmeaza corpul metodei ...
```

metoda care **rescrie** (***overrides***) codul metodei cu acelasi nume din clasa extinsa (in acest caz clasa **`Object`**).

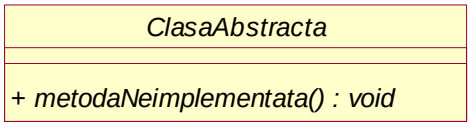
Dupa adaugarea acestei metode, apelul **`equals()` va conduce la executia noului cod**, pe cand apelul **`super.equals()`duce la executia codului din clasa extinsa** (in acest caz codul din **`Object`**).

OC.2.5.5. Clase si operatii (metode) abstracte

Simbolul UML pentru o **metoda abstracta** (declarata **`abstract`**) prevede scrierea numelui ei **inclinat** (font *Italic*). **Clasele abstracte sunt reprezentate in UML prin nume scris inclinat** (*Italic*).



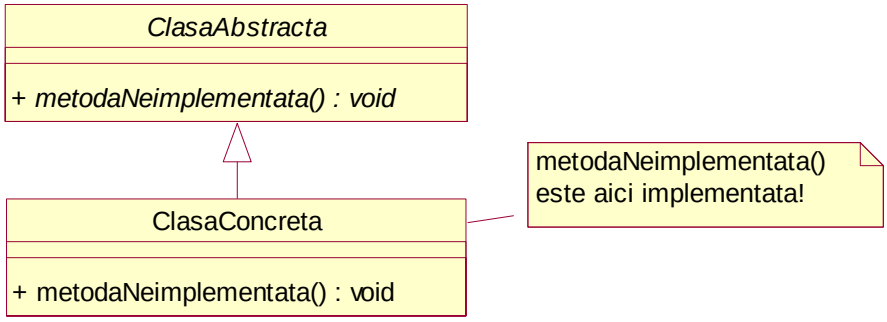
Orice clasa Java care contine cel putin o metoda abstracta trebuie neaparat declarata `abstract` (in caz contrar nu poate fi compilata!).



Codul Java corespunzator diagramei UML de mai sus este urmatorul:

```
1  public abstract class ClasaAbstracta {
2    public abstract void metodaNeimplementata();
3  }
```

Pentru a se putea crea obiecte avand ca tip o clasa abstracta, ea **trebuie extinsa prin mostenire**, si **toate metodele ei neimplementate trebuie implementate in subclasa concreta** respectiva.

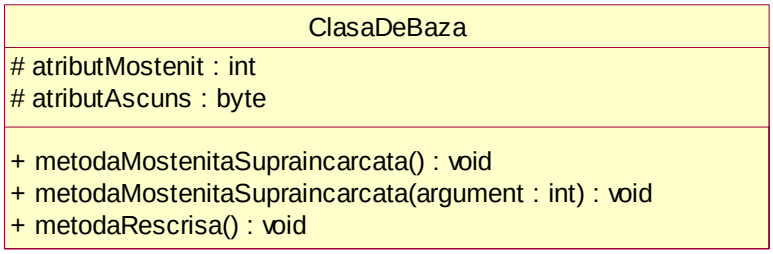


Codul Java corespunzator diagramei UML de mai sus este urmatorul:

```
1  public class ClasaConcreta extends ClasaAbstracta {
2
3    public void metodaNeimplementata() {
4      // metoda trebuie sa nu fie abstracta pentru a nu fi abstracta si clasa
5    }
6
7  }
```

Exemplul urmator **ilustreaza situatiile de mostenire, ascundere si adaugare de atribute, si de mostenire, rescriere si adaugare de metode, in subclase**.

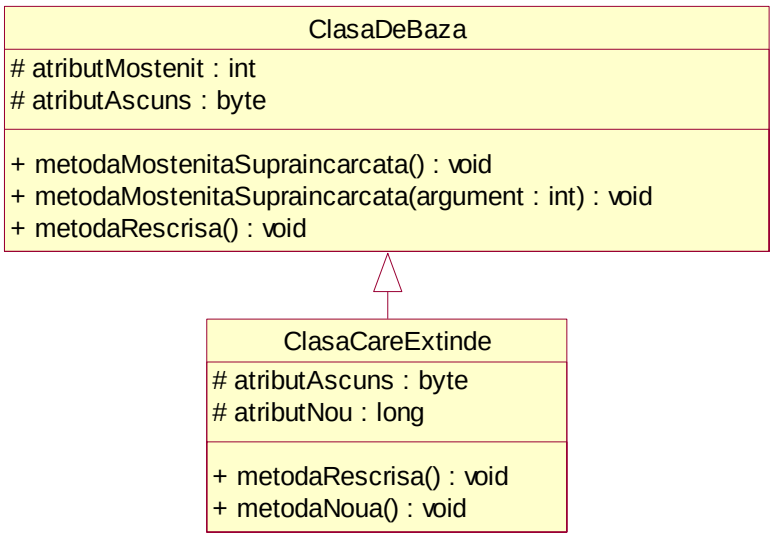
Diagramei UML:



ii corespunde codul Java:

```
1  public class ClasaDeBaza {
2    protected int atributMostenit;    // atribut partajat cu subclasa
3    protected byte atributAscuns;     // atribut corespunzator clasei de baza
4
5    public void metodaMostenitaSupraincarcata() {
6      // implementare corespunzatoare lipsei parametrilor
7    }
8
9    public void metodaMostenitaSupraincarcata(int argument) {
10     // implementare corespunzatoare parametrului de tip int
11   }
12
13   public void metodaRescrisa() {
14     // implementare de baza
15   }
16 }
```

Diagramei UML:



ii corespunde codul Java:

```
 1  public class ClasaCareExtinde extends ClasaDeBaza {
 2    protected byte atributAscuns;     // atribut corespunzator subclasei
 3    protected long atributNou;        // atribut nou, nepartajat cu clasa de baza
 4
 5    public void metodaRescrisa() {
 6      // reimplementare (rescriere a codului)
 7    }
 8
 9    public void metodaNoua() {
10      // metoda noua, nepartajata cu clasa de baza
11    }
12  }
```

Urmatorul cod Java **ilustreaza modul de utilizare a atributelor si metodelor de mai sus**.

```
 1  Public class UtilizareClase {
 2
 3    public static void main(String[] args) {
 4
 5      ClasaDeBaza obiectDeBaza = new ClasaDeBaza();     // clasa de baza (extinsa)
 6
 7      obiectDeBaza.atributMostenit    // utilizarea atributului partajat cu subclasa
 8
 9      obiectDeBaza.atributAscuns      // utilizarea atributului din clasa de baza
10
11
12      // apelul metodei din clasa de baza (corespunzatoare lipsei parametrilor)
13      obiectDeBaza.metodaMostenitaSupraincarcata()
14
15      // apelul metodei din clasa de baza (corespunzatoare parametrului tip int)
16      obiectDeBaza.metodaMostenitaSupraincarcata(1000)
17
18      // apelul metodei din clasa de baza (implementare de baza)
19      obiectDeBaza.metodaRescrisa()
20
21
22      ClasaCareExtinde obiectExtins = new ClasaCareExtinde();         // subclasa
23
24      obiectExtins.atributNou         // utilizarea atributului nou
25
26      obiectExtins.atributMostenit    // utilizarea atributului partajat
27
28      obiectExtins.atributAscuns      // utilizarea atributului din subclasa
29
30      super.atributAscuns             // utilizarea atributului din clasa de baza
31
32
33      // apelul metodei noi din subclasa
34      obiectExtins.metodaNoua()
35
36      // apelul metodei din clasa de baza (corespunzatoare lipsei parametrilor)
37      obiectExtins.metodaMostenitaSupraincarcata()
38
39      // apelul metodei din clasa de baza (corespunzatoare parametrului tip int)
40      obiectExtins.metodaMostenitaSupraincarcata(1000)
41
42      // apelul metodei din subclasa (implementarea noua, rescrisa)
43      obiectExtins.metodaRescrisa()
44
45      // apelul metodei din clasa de baza (implementare de baza)
46      super.metodaRescrisa()
47    }
58  }
```

Sa reluam **exemplul claselor care incapsuleaza informatiile privind numere complexe**.

Vom considera o **clasa Java abstracta** (din care **nu pot fi create in mod direct instante, obiecte**), **`ComplexAbstract`**, **care contine partea comuna a interfetei publice a claselor concrete** (din care pot fi create in mod direct instante, obiecte) care reprezinta numerele complexe.

```
1  // Clasa abstracta (din care nu pot fi create direct obiecte)
2  // care reprezinta partea comuna a interfetei publice a claselor
3  // ComplexPolar si ComplexCartezian
4
5  public abstract class ComplexAbstract {
6
7    // Metode abstracte, neimplementate
8    public abstract double getReal();
9    public abstract double getImag();
10   public abstract double getModul();
11   public abstract double getFaza();
12 }
```

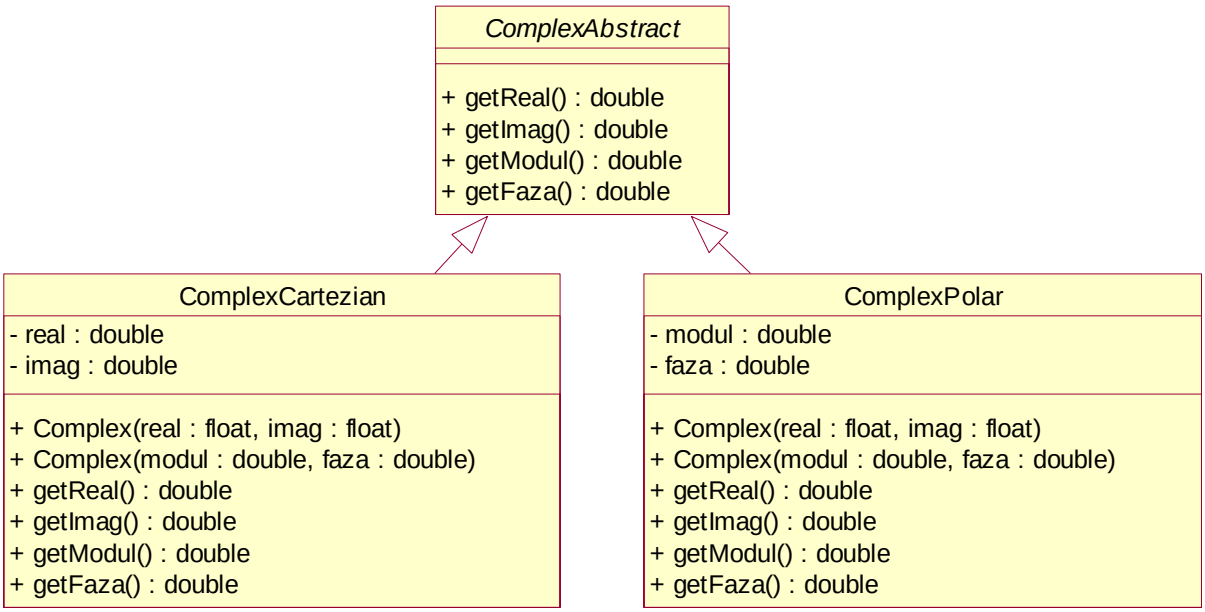
Clasa abstracta **`ComplexAbstract`** poate fi extinsa prin mostenire (**`extends`**) de **o clasa concreta care reprezinta numerele complexe in format cartezian**, **`ComplexCartezian`**.

```
1  // Clasa concreta, din care pot fi create direct obiecte, care extinde
2  // (mosteneste) clasa abstracta ComplexAbstract
3  public class ComplexCartezian extends ComplexAbstract {
4
5    // Atribute private (ascunse, inaccesibile din exteriorul clasei)
6    private double real;
7    private double imag;
8
9    public void Complex(float real, float imag) {
10       this.real = real;
11       this.imag = imag;
12   }
13   public void Complex(double modul, double faza) {
14       this.real = modul * Math.cos(faza);
15       this.imag = modul * Math.sin(faza);
16   }
17   public double getReal() {
18    return this.real;
19   }
20   public double getImag() {
21    return this.imag;
22   }
23   public double getModul() {
24    return Math.sqrt(this.real*this.real + this.imag*this.imag);
25   }
26   public double getFaza() {
27    return Math.atan2(this.real, this.imag);
28   }
29 }
```

Clasa abstracta **`ComplexAbstract`** poate fi extinsa prin mostenire si de **o clasa concreta care reprezinta numerele complexe in format polar**, **`ComplexPolar`**.

```
1  // Clasa concreta, din care pot fi create direct obiecte, care extinde
2  // (mosteneste) clasa abstracta ComplexAbstract
3  public class ComplexPolar extends ComplexAbstract {
4
5    // Atribute private (ascunse, inaccesibile din exteriorul clasei)
6    private double modul;
7    private double faza;
8
9    public void Complex(float real, float imag) {
10       this.modul = Math.sqrt(real*real + imag*imag);
11       this.faza = Math.atan2(real, imag);
12   }
13   public void Complex(double modul, double faza) {
14       this.modul = modul;
15       this.faza = modul;
16   }
17   public double getReal() {
18    return this.modul*Math.cos(this.faza);
19   }
20   public double getImag() {
21    return this.modul*Math.sin(this.faza);
22   }
23   public double getModul() {
24    return this.modul;
25   }
26   public double getFaza() {
27    return this.faza;
28   }
29 }
```

Diagrama UML echivalenta:



OC.2.5.6. Mostenirea multipla. Interfetele Java si implementarea lor

Moştenirea multiplă

- poate conduce la **conflicte (coliziuni) de nume**, fapt pentru care **Ada95 şi Java nu oferă moştenire multiplă**,
- **nu trebuie să fie modalitatea prin care să se realizeze o fuziune între două ansambluri de clase construite în mod independent**,
- **utilizarea ei trebuie anticipată**.

Moştenirea nu este o necesitate absolută, ea putând fi **înlocuită întotdeauna prin delegare**.

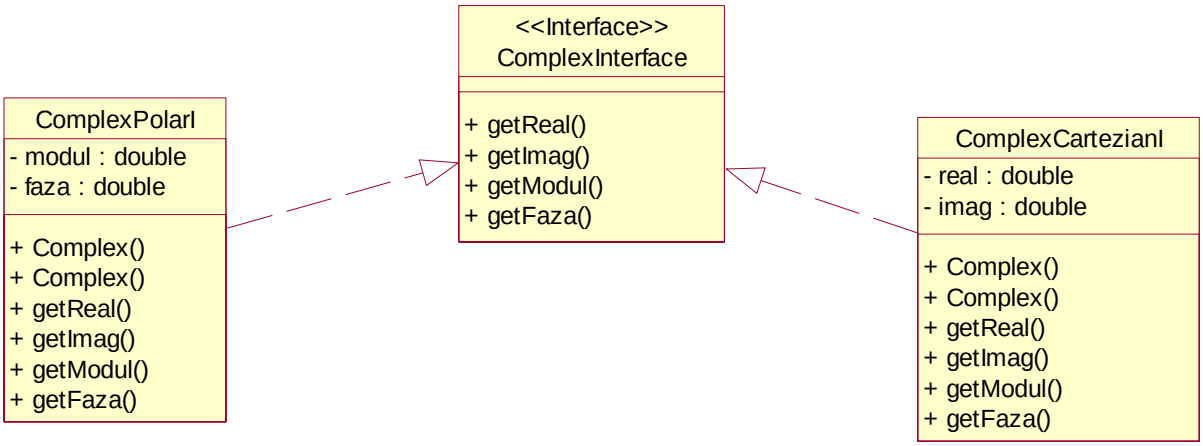
Implementarea interfetelor este **alternativa Java la mostenirea multipla** (extinderea mai multor clase), deoarece in Java mostenirea multipla nu este permisa.

In Java, dar si in UML, exista notiunea de **interfata**, care semnifica un **caz particular de clasa**, care **poate contine doar metode** (nu si atribute, ci doar cel mult variabile finale – constante) **abstracte (neimplementate)**.

Astfel, clasa **`ComplexAbstract`** ar putea fi inlocuita cu o interfata **`ComplexInterface`**:

```
 1  // Interfata (colectie de metode neimplementate)
 2  // care reprezinta un contract privind interfata publica
 3
 4  public interface ComplexInterface {
 5
 6    // Metode abstracte, neimplementate
 7    public abstract double getReal();
 8    public abstract double getImag();
 9    public abstract double getModul();
10    public abstract double getFaza();
11  }
```

Reprezentarea in UML este (se observa **linia intrerupta** care **semnifica implementarea unei interfete**, spre deosebire de cea continua care semnifica extinderea unei clase):



Interfata reprezinta un **contract privind setul de metode pe care trebuie sa le implementeze clasele concrete** care implementeaza (concretizeaza) interfata.

Interfata **`ComplexInterface`** poate fi **concretizata, implementata** (**`implements`**), **de o clasa concreta care reprezinta numerele complexe in format cartezian**, **`ComplexCartezianI`**.

```
1  // Clasa concreta, din care pot fi create direct obiecte, care
2  // implementeaza (concretizeaza) interfata ComplexInterface
3  public class ComplexCartezianI implements ComplexInterface {
4
5    // Atribute private (ascunse, inaccesibile din exteriorul clasei)
6    private double real;
7    private double imag;
8
9    public void Complex(float real, float imag) {
10       this.real = real;
11       this.imag = imag;
12   }
13   public void Complex(double modul, double faza) {
14       this.real = modul * Math.cos(faza);
15       this.imag = modul * Math.sin(faza);
16   }
17   public double getReal() {
18    return this.real;
19   }
20   public double getImag() {
21    return this.imag;
22   }
23   public double getModul() {
24    return Math.sqrt(this.real*this.real + this.imag*this.imag);
25   }
26   public double getFaza() {
27    return Math.atan2(this.real, this.imag);
28   }
29 }
```

Interfata **ComplexInterface** poate fi **concretizata, implementata, si de o clasa concreta care reprezinta numerele complexe in format polar**, **ComplexPolarI**.

```
1  // Clasa concreta, din care pot fi create direct obiecte, care
2  // implementeaza (concretizeaza) interfata ComplexInterface
3  public class ComplexPolarI implements ComplexInterface {
4
5    // Atribute private (ascunse, inaccesibile din exteriorul clasei)
6    private double modul;
7    private double faza;
8
9    public void Complex(float real, float imag) {
10       this.modul = Math.sqrt(real*real + imag*imag);
11       this.faza = Math.atan2(real, imag);
12   }
13   public void Complex(double modul, double faza) {
14       this.modul = modul;
15       this.faza = modul;
16   }
17   public double getReal() {
18    return this.modul*Math.cos(this.faza);
19   }
20   public double getImag() {
21    return this.modul*Math.sin(this.faza);
22   }
23   public double getModul() {
24    return this.modul;
25   }
26   public double getFaza() {
27    return this.faza;
28   }
29 }
```

Un alt bun exemplu de **implementare a unei interfete** este cel al unei stive. **Interfata specifica** acele **metode pe care trebuie sa le implementeze o clasa pentru a avea comportamentul dorit** (de stiva in acest caz).

```
1  public interface StackInterface {
2    boolean empty();
3    void push( Object x);
4    Object pop() throws EmptyStackException;
5    Object peek() throws EmptyStackException;
6  }
```

Iata **codul unei clase stiva** (**`Stack`**) **care implementeaza interfata** de mai sus, **utilizand un obiect lista inlantuita** (**`Vector`**) sub forma unui **atribut privat (inaccesibil vreunui cod extern)**.

```
1  public class Stack implements StackInterface {
2    private Vector v = new Vector(); // utilizeaza clasa java.util.Vector
3
4    public void push(Object item) { v.addElement(item); }
5    public Object pop() {
6      Object obj = peek();
7      v.removeElementAt(v.size() - 1);
8      return obj;
9    }
10   public Object peek() throws EmptyStackException {
11     if (v.size() == 0) throw new EmptyStackException();
12     return v.elementAt(v.size() - 1);
13   }
14   public boolean empty() { return v.size() == 0; }
15 }
```

Iata si **codul unei clase stiva** (**`Stack`**) care **implementeaza interfata** de mai sus si **extinde clasa lista inlantuita** (**`Vector`**).

```
1  public class Stack extends Vector implements StackInterface {
2    public Object push(Object item) { addElement(item); return item; }
3    public Object pop() {
4      Object obj;
5      int len = size();
6      obj = peek();
7      removeElementAt( len - 1);
8      return obj;
9    }
10   public Object peek() {
11     int len = size();
12     if (len == 0) throw new EmptyStackException();
13     return elementAt( len - 1);
14   }
15   public boolean empty() { return size() == 0;}
16 }
```

Toate metodele publice ale clasei **`Vector`**, **printre care si metode de inserare**, **pot fi invocate pentru obiectele clasei stiva in aceasta implementare**. In acest fel, pe langa comportamentul tipic stivei, **utilizatorii pot declansa comportamente atipice** (invocand metode de inserare, etc.), **care incalca principiul de functionare**.

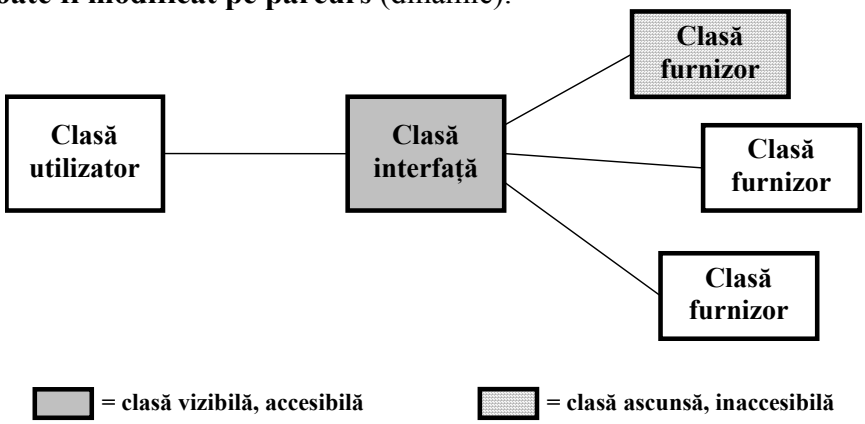
De exemplu, codul:

```
Vector v = new Stack();  // cod legal – referinta la clasa de baza
                         // poate fi initializata cu obiect din subclasa
v.insertElementAt(x, 2); // cod legal – dar inserarea unui obiect in stiva
                         // incalca principiul de functionare al stivei
```

OC.2.5.7. Delegarea ca alternativa a mostenirii

O **alternativa la mostenire** este **delegarea unor sarcini catre obiecte ale altor clase**. Intre clasa care deleaga sarcinile si clasa care le ofera exista **relatia de compunere**.

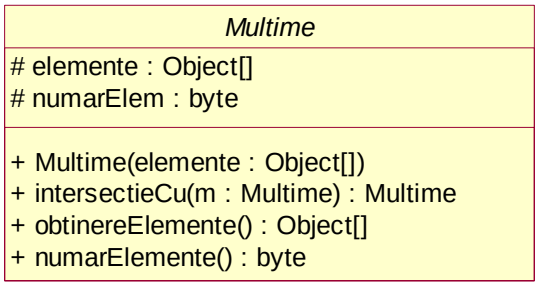
Delegarea unor sarcini catre obiecte ale altor clase poate reduce cuplajul în model:
- clientul (utilizatorul) **nu cunoaşte direct** furnizorul,
- furnizorul **poate fi modificat pe parcurs** (dinamic).



Exemplul urmator **ilustreaza** mai intai **mostenirea, ascunderea, rescrierea si adaugarea de membri** (atribute si metode) **in subclase**, ca **si o parte a elementelor speciale utilizate in declararea claselor, atributelor si metodelor**.

Apoi sunt **ilustrate doua moduri de reutilizare a codului**:
- **prin mostenire** (extindere),
- **prin delegare** (comunere).

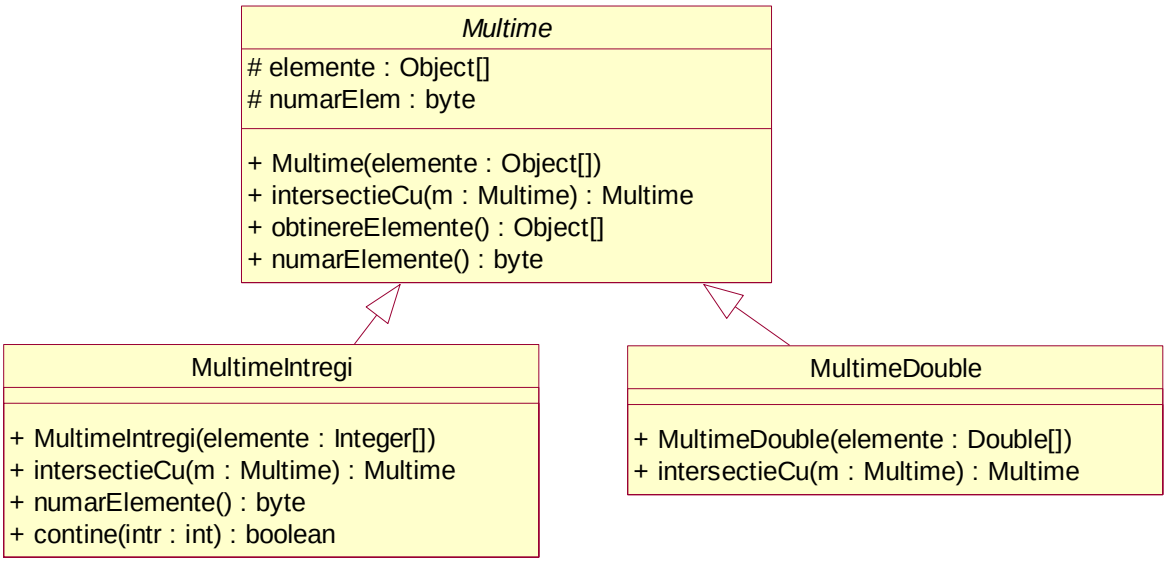
Diagramei UML:



ii corespunde codul Java:

```
 1  public abstract class Multime {                    // clasa declarata abstract
 2    protected Object[] elemente;
 3    protected byte numarElem;
 4
 5    public Multime(Object[] elemente) {       // parametru generic tip Object[]
 6        this.elemente = elemente;             // acces la obiectul curent cu this
 7        numarElem = (byte) elemente.length;   // conversie de tip de la int la byte
 8    }
 9
10    public abstract Multime intersectieCu(Multime m);  // metoda declarata abstract
11                                        // valoare returnata generica tip Multime
12
13    public Object[] obtinereElemente() { // valoare returnata generica tip Object[]
14        return elemente;
15    }
16
17    public byte numarElemente() {        // implementare de baza
18        return numarElem;
19    }
20  }
```

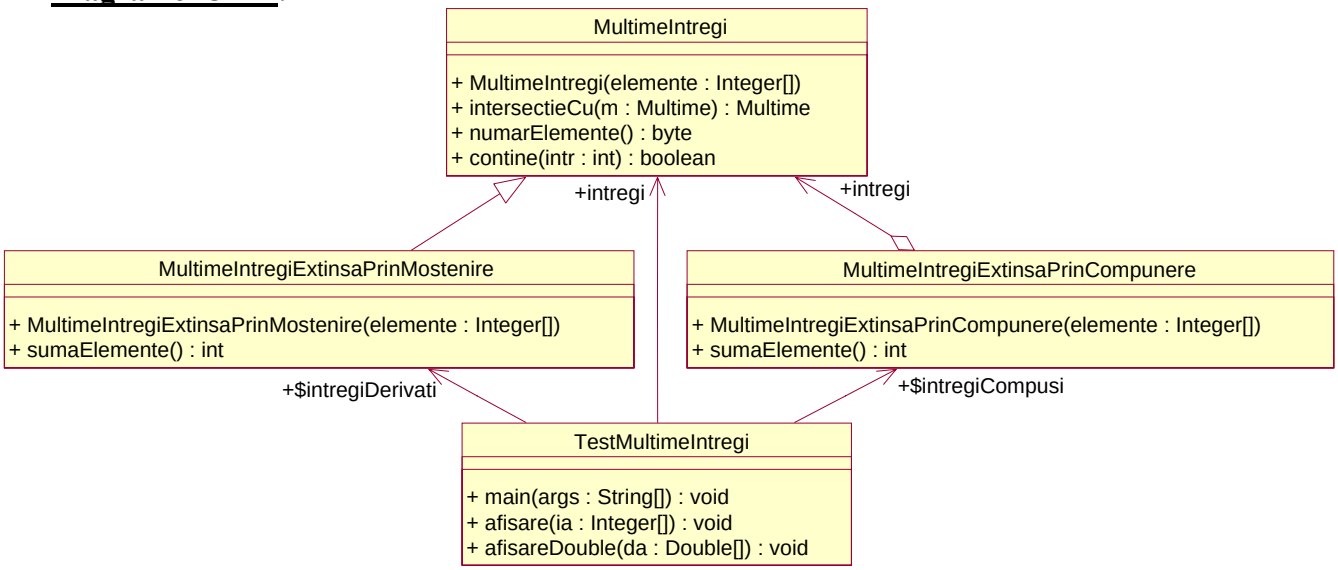
Diagramei UML:



ii corespunde codul Java:

```
 1  public class MultimeIntregi extends Multime {
 2
 3    public MultimeIntregi(Integer[] elemente) { // parametru concret tip Integer[]
 4        super(elemente);  // apelul constructorului clasei de baza Multime cu super
 5    }
 6
 7    public final Multime intersectieCu(Multime m) { // implementarea metodei
 8                                        // declarata abstract in clasa de baza
 9        Multime mNoua;
10        Integer[] elementeIntersectie;
11        int nrElemente = 0;
12
13        for (int i=0; i< elemente.length; i++) {
14          for (int j=0; j< m.elemente.length; j++) {
15            if (elemente[i].equals(m.elemente[j])) {
16              nrElemente++;
17            }
18          }
19        }
20        int index = 0;
21        elementeIntersectie = new Integer[nrElemente];
22        for (int i=0; i< elemente.length; i++) {
23          for (int j=0; j< m.elemente.length; j++) {
24            if (elemente[i].equals(m.elemente[j])) {
25              elementeIntersectie[index++] = new Integer(elemente[i].toString());
26            }
27          }
28        }
29        mNoua = new MultimeIntregi(elementeIntersectie);
30
31        return mNoua;
32    }
33
34    public byte numarElemente() {          // reimplementare (rescriere cod)
35        return (byte) elemente.length;     // conversie de tip de la int la byte
36    }
37
38    public boolean contine(int intr) {     // metoda noua
39        for (int i=0; i< elemente.length; i++) {
40          Integer inte = (Integer) elemente[i];
41          if (inte.intValue() == intr) {
42            return true;
43          }
44        }
45        return false;
46    }
47  }
```

Diagramei UML:



ii corespund codurile Java:

- ale unei clase care extinde prin mostenire (extindere):

```
1  public class MultimeIntregiExtinsaPrinMostenire extends MultimeIntregi {
2
3    public MultimeIntregiExtinsaPrinMostenire(Integer[] elemente) {
4        super(elemente);
5    }
6
7    public int sumaElemente() {     // metoda noua
8        int suma = 0;
9        Integer[] ti = (Integer[]) elemente;   // utilizare atribut mostenit elemente
10       for (int i=0; i< ti.length; i++) {
11         suma = suma + ti[i].intValue();
12       }
13       return suma;
14   }
15 }
```

- ale unei clase care extinde prin delegare (compunere):

```
1  public class MultimeIntregiExtinsaPrinCompunere {
2    public MultimeIntregi intregi;                  // obiect componenta
3
4    public MultimeIntregiExtinsaPrinCompunere(Integer[] elemente) {
5        intregi = new MultimeIntregi(elemente);
6    }
7
8    public int sumaElemente() {     // metoda noua
9        int suma = 0;
10       Integer[] ti = (Integer[]) intregi.obtinereElemente();
11       for (int i=0; i< ti.length; i++) {
12         suma = suma + ti[i].intValue();
13       }
14       return suma;
15   }
   }
```

- ale unei clase care permite testarea comportamentului (si compararea modului de utilizare) in cele doua cazuri:

```
 1  public class TestMultimeIntregi {
 2    public MultimeIntregi intregi;
 3    public static MultimeIntregiExtinsaPrinMostenire intregiDerivati;
 4    public static MultimeIntregiExtinsaPrinCompunere intregiCompusi;
 5
 6    public static void main(String[] args) {
 7      int i;
 8
 9      Integer[] tablouA = { new Integer(1), new Integer(3), new Integer(5) };
10      MultimeIntregi multimeA = new MultimeIntregi(tablouA);
11
12      intregiCompusi = new MultimeIntregiExtinsaPrinCompunere(tablouA);
13      int suma = intregiCompusi.sumaElemente();
14      System.out.println("Suma elementelor " + suma);
15
16      intregiDerivati = new MultimeIntregiExtinsaPrinMostenire(tablouA);
17      suma = intregiDerivati.sumaElemente();
18      System.out.println("Suma elementelor " + suma);
19
20      i = 2;
21      if (multimeA.contine(i))
22        System.out.println("Multimea contine " + i);
23      else
24        System.out.println("Multimea nu contine " + i);
25
26      Integer[] tabloulB = { new Integer(2), new Integer(3), new Integer(4) };
27      MultimeIntregi multimeaB = new MultimeIntregi(tabloulB);
28
29      MultimeIntregi intersectiaAcuB =
30         (MultimeIntregi) multimeA.intersectieCu(multimeaB); // necesara conversie
31      Integer[] tabloulAB =
32         (Integer[]) intersectiaAcuB.obtinereElemente();     // necesara conversie
33
34      Double[] tabloulD = { new Double(1.1), new Double(3.3), new Double(5.5) };
35      MultimeDouble multimeaD = new MultimeDouble(tabloulD);
36    }
37  }
```

OC.2.5.8. Posibilitatile oferite de mostenire

Subclasele (clasele care extind prin mostenire) pot sa:

- mareasca gradul de detaliere al obiectelor:
  - **adaugand noi atribute**, inexistente in clasa de baza, ceea ce inseamna introducerea unui **grad mai inalt de detaliere a starilor** obiectelor din subclasa fata de cele din clasa de baza,
  - **adaugand noi metode**, inexistente in clasa de baza, ceea ce inseamna introducerea unui **grad mai inalt de detaliere a comportamentului** obiectelor din subclasa fata de cele din clasa de baza,

- mareasca gradul de concretete a obiectelor:
  - **implementand eventualele metode abstracte** din clasa de baza, ceea ce inseamna un **grad mai mare de concretete a comportamentului obiectelor** din subclasa fata de cele din clasa de baza,

- introduca diferentieri ale obiectelor:
  - **redeclarand unele dintre atributele existente** in clasa de baza (schimbandu-le tipul), ceea ce inseamna **ascunderea** (*hiding*) atributelor cu acelasi nume din clasa de baza, adica introducerea unei **diferentieri a starii obiectelor din subclasa** fata de cele din clasa de baza,
  - **reimplementand unele dintre metodele existente** in clasa de baza, ceea ce inseamna **rescrierea** (*overriding*) metodelor cu acelasi nume din clasa de baza, adica introducerea unei **diferentieri a comportamentului obiectelor din subclasa** fata de cele din clasa de baza.

OC.2.5.9. Elemente care pot fi si elemente care nu pot fi mostenite

Subclasele mostenesc toate:
- **atributele din clasa de baza care nu sunt ascunse** prin redeclarare,
- **metodele din clasa de baza care nu sunt rescrise** prin reimplementare.

Altfel spus, **obiectele din subclasa poseda toate atributele declarate in clasa de baza**, si **pot utiliza toate metodele declarate in clasa de baza**. Aceasta reutilizare a codului clasei de baza de catre subclase este principalul beneficiu al utilizarii extinderii prin mostenire.

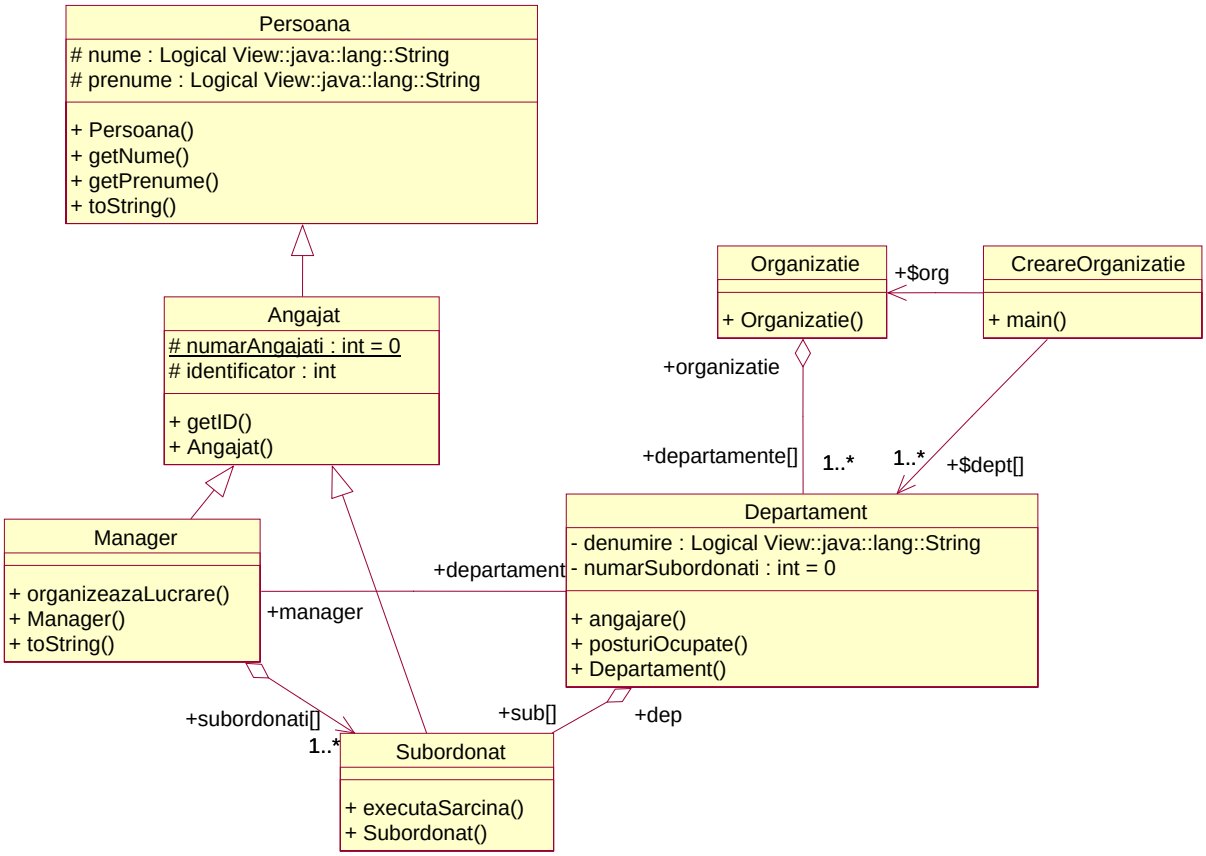
Subclasele nu mostenesc constructorii clasei de baza (**au constructori proprii**), dar **pot face apel la constructorii clasei de baza** (daca se intampla acest lucru, apelul la constructorul clasei de baza, realizat prin apelul **`super()`**, trebuie sa fie prima declaratie din corpul constructorului subclasei).

Subclasele nu mostenesc nici:
- **atributele cu caracter global** (static),
- **metodele cu caracter global** (static).

Altfel spus, **membrii globali** (statici) **ai clasei de baza nu pot fi mosteniti** de obiectele din subclasa (**ei tin strict de clasa in care au fost declarati**).

OC.2.5.10. Exemplu complex

In continuare vor fi **exemplificate in UML si Java cateva dintre aspectele discutate anterior. Diagrama de clase** de mai jos, **corespunzatoare codului Java care urmeaza**, este un **exemplu de model UML al relatiilor dintre clase si al ierarhiilor de clase obtinute prin mostenire**.



Clasa de baza a ierarhiei persoanelor (care modeleaza identitatea unei persoane):

```
 1  public class Persoana {
 2    protected String nume;
 3    protected String prenume;
 4
 5    public Persoana(String nume, String prenume) {
 6      this.nume = nume;         // this = referinta la obiectul curent
 7      this.prenume = prenume;
 8      System.out.println("A fost creata persoana " + this.toString());
 9    }
10
11    public String getNume() {
12      return this.nume;
13    }
14
15    public String getPrenume() {
16      return this.prenume;
17    }
18
19    public String toString() {
20     return (this.nume + " " + this.prenume);
21    }
22  }
```

Atributele **nume** si **prenume** sunt **protected**, adica sunt accesibile doar in clasa **Persoana**, in subclasele acesteia si in clasele care se afla in acelasi director (pachet) cu clasa **Persoana**.

Cuvantul cheie **this** este **referinta la obiectul curent**.

Metodele publice **getNume()** si **getPrenume()** ofera tuturor claselor externe posibilitatea de a obtine valoarea atributelor. Clasele aflate in alte directoare nu pot insa modifica aceste atribute.

Metoda publica **toString()** ofera claselor externe posibilitatea de a obtine continutul unui obiect de tip **Persoana**, sub forma unui sir de caractere reprezentand concatenarea atributelor obiectului.

O clasa care extinde clasa **Persoana**, modeland informatiile privind un angajat al unei organizatii (modelata de clasa **Organizatie**), este clasa **Angajat**.

```
 1  public class Angajat extends Persoana {
 2    protected static int numarAngajati = 0;
 3    protected int identificator;
 4    protected Organizatie organizatie;
 5
 6    public Angajat(String nume, String prenume, Organizatie organizatie) {
 7      super(nume, prenume);    // apelul constructorului superclasei
 8
 9      this.numarAngajati++;    // echivalent cu:  Angajat.numarAngajati++;
10
11      this.identificator = numarAngajati;
12      System.out.println("A fost creat angajatul cu ID " + identificator);
13    }
14
15    public int getID() {
16      return this.identificator;
17    }
18  }
```

Codul clasei `Organizatie` este urmatorul.

```
 1  public class Organizatie {
 2    public Departament departamente[];
 3
 4    public Organizatie(int numarDepartamente) {
 5
 6      // Creare structura departamente
 7      this.departamente = new Departament[numarDepartamente];
 8
 9      System.out.println("A fost creata structura organizatiei");
10    }
11  }
```

`Angajat` adauga un atribut cu caracter **`protected`** si **global** (**`static`**), **`numarAngajati`**, si alte doua atribute cu caracter **`protected`**.

De fiecare data cand se creaza un nou obiect de tip **`Angajat`**, **constructorul incrementeaza atributul global** **`numarAngajati`**, care este folosit **pentru a contoriza numarul de obiecte** de tip **`Angajat`** create.

Desigur, constructorul initializeaza atributele **`non-statice`** (acelea care sunt variabile distincte pentru fiecare obiect). Printre ele, **atributul `identificator` foloseste informatia actualizata a atributului** global **`numarAngajati` pentru a aloca un cod unic fiecarui obiect** de tip **`Angajat`**.

Apelul **`super()`** reprezinta **apelul unui constructor al clasei de baza (supraclasei), `Persoana`**.

Atat atributele **`nume`** si **`prenume`** cat si metodele **`toString()`**, **`getNume()`** si **`getPrenume()`** sunt **reutilizate** in codul clasei **`Angajat`**. Altfel spus, un obiect al clasei **`Angajat`** are si acele atribute si metode, pe langa cele suplimentar declarate chiar in clasa **`Angajat`**.

Atributele **`nume`** si **`prenume`** si metodele **`toString()`**, **`getNume()`** si **`getPrenume()`** reprezinta **codul comun tuturor obiectelor** de tip **`Persoana`**, inclusiv al obiectelor de tip **`Angajat`**. Clasa **`Angajat`** este o specializare a clasei **`Persoana`**, obiectele de tip **`Angajat`** avand in plus (fata de cele de tip simplu **`Persoana`**) trei atribute si o metoda.

O clasa care extinde clasa **`Angajat`**, modeland informatiile privind un manager de department (modelat in clasa **`Departament`**) din cadrul unei organizatii, este clasa **`Manager`**.

```
 1  public class Manager extends Angajat {
 2    public Subordonat subordonati[];
 3    public Departament departament;
 4
 5    public Manager(Persoana persoana, Organizatie organizatie) {
 6      super(persoana.nume, persoana.prenume, organizatie);
 7
 8      // se foloseste noul cod al metodei toString()
 9      System.out.println("A fost creat un manager, " + this.toString());
10    }
11    public void organizeazaLucrare(int numarZile) {
12      this.subordonati = departament.sub;
13      int orePePersoana = numarZile * 8 / subordonati.length;
14
15      for (int no=0; no < subordonati.length ; no++) {
16        subordonati[no].executaSarcina(orePePersoana);
17      }
18    }
19    // rescrierea codului (reimplementarea) metodei toString()
20    public String toString() {
21      // cu super.toString() se apeleaza codul metodei din clasa Persoana
22      // astfel incat noul cod este o specializare a codului initial
23      return (super.toString() + ", Manager de Departament");
24    }
25  }
```

Atributele care privesc relatiile cu organizatia (**`departament`**) si subordonatii (**`subordonati`**), si metodele care modeleaza activitatile unui manager (**`organizeazaLucrare()`**) specializeaza obiectele de tip **`Manager`**.

Codul metodei `toString()` este **rescris**, dar el **apeleaza codul original**, ceea ce constituie atat o **specializare** cat si o **reutilizare**.

O clasa care extinde clasa **`Angajat`**, modeland informatiile privind unui simplu membru al unui department din cadrul unei organizatii, este clasa **`Subordonat`**.

```
 1  public class Subordonat extends Angajat {
 2    public Departament dep;
 3
 4    public Subordonat(Persoana persoana, Organizatie organizatie,
 5                                                Departament departament) {
 6      super(persoana.nume, persoana.prenume, organizatie);
 7
 8      this.dep = departament;
 9      System.out.println("A fost creat un subordonat, " + this.toString());
10    }
11
12    public void executaSarcina(int numarOre) {
13      System.out.println("Angajatul " + nume + " " + prenume + " (ID = " +
14          identificator + ") a executat sarcina in " + numarOre + " ore");
15    }
16  }
```

Atributele care privesc relatiile cu organizatia (**`dep`**) si metodele care modeleaza activitatile unui simplu membru (**`executaSarcina()`**) specializeaza obiectele de tip **`Subordonat`**.

Codul clasei **`Departament`** este urmatorul.

```
 1  public class Departament {
 2    private String denumire;
 3    private int numarSubordonati = 0;
 4    public Organizatie organizatie;
 5    public Subordonat sub[];
 6    public Manager manager;
 7
 8    public Departament(String denumire, Organizatie organizatie,
 9                            Manager manager, int numarSubordonati) {
10      // Stabilirea denumirii
11      this.denumire = denumire;
12
13      // Crearea legaturii cu organizatia
14      this.organizatie = organizatie;
15
16      // Crearea legaturilor cu managerul
17      // (navigabilitatea asocierii este bidirectionala)
18      this.manager = manager;
19      manager.departament = this;
20
21      // Crearea structurii subordonatilor
22      this.sub = new Subordonat[numarSubordonati];
23
24      System.out.println("A fost creata structura dept. " + denumire);
25    }
26
27    public void angajare(Persoana persoana) {
28      sub[numarSubordonati++] = new Subordonat(persoana, organizatie, this);
29    }
30    public boolean posturiOcupate() {
31      return (numarSubordonati == sub.length);
32    }
33  }
```

Codul clasei **`CreareOrganizatie`**, care ilustreaza **modul de initializare al obiectelor din clasele anterioare**, este urmatorul.

```
 1  public class CreareOrganizatie {
 2    public static Organizatie org;
 3    public static Departament dept[];
 4
 5    public static void main(java.lang.String[] args) {
 6      Persoana persoana;
 7      Manager manager;
 8
 9      // Creare organizatie
10      org = new Organizatie(4);  // 4 departamente
11
12      // Creare departamente
13      dept = org.departamente;
14
15      persoana = new Persoana("G.", "N.T.");
16      // Persoana e angajata si devine subordonat in departament Personal
17      manager = new Manager(persoana, org);
18
19      // Creare departament Personal
20      dept[0] = new Departament("Personal", org, manager, 3);
21
22      // Creare persoana
23      persoana = new Persoana("F.", "L.");
24
25      // Persoana e angajata si devine subordonat in departament Personal
26      if (!dept[0].posturiOcupate())    dept[0].angajare(persoana);
27
28      persoana = new Persoana("T.", "N.");
29
30      // Persoana e angajata si devine subordonat in departament Personal
31      if (!dept[0].posturiOcupate())    dept[0].angajare(persoana);
32
33      persoana = new Persoana("H.", "L.");
34
35      // Persoana e angajata si devine subordonat in departament Personal
36      if (!dept[0].posturiOcupate())    dept[0].angajare(persoana);
37
38      // O noua persoana nu poate fi angajata in departament Personal
39      if (!dept[0].posturiOcupate())    dept[0].angajare(persoana);
40
41      // Creare departament Tehnic
42      persoana = new Persoana("W.", "D.");
43      manager = new Manager(persoana, org);
44      dept[1] = new Departament("Tehnic", org, manager, 4);
45
46      // Creare departament Vanzari
47      persoana = new Persoana("E.", "V.");
48      manager = new Manager(persoana, org);
49      dept[2] = new Departament("Vanzari", org, manager, 7);
50
51      // Creare departament Financiar
52      persoana = new Persoana("R.", "M.");
53      manager = new Manager(persoana, org);
54      dept[3] = new Departament("Financiar", org, manager, 4);
55
56      System.out.println("A fost creata organizatia");
57    }
58  }
```

Efectul executiei programului **`CreareOrganizatie`**:

```
05_ISw\ISw04_Curs\Persoana>java CreareOrganizatie
A fost creata structura organizatiei

A fost creata persoana G. N.T.
A fost creata persoana G. N.T., Manager de Departament
A fost creat angajatul cu ID 1
A fost creat un manager, G. N.T., Manager de Departament
A fost creata structura departamentului Personal
A fost creata persoana F. L.
A fost creata persoana F. L.
A fost creat angajatul cu ID 2
A fost creat un subordonat, F. L.
A fost creata persoana T. N.
A fost creata persoana T. N.
A fost creat angajatul cu ID 3
A fost creat un subordonat, T. N.
A fost creata persoana H. L.
A fost creata persoana H. L.
A fost creat angajatul cu ID 4
A fost creat un subordonat, H. L.
G. N.T., Manager de Departament
A fost creata persoana W. D.
A fost creata persoana W. D., Manager de Departament
A fost creat angajatul cu ID 5
A fost creat un manager, W. D., Manager de Departament
A fost creata structura departamentului Tehnic
A fost creata persoana E. V.
A fost creata persoana E. V., Manager de Departament
A fost creat angajatul cu ID 6
A fost creat un manager, E. V., Manager de Departament
A fost creata structura departamentului Vanzari
A fost creata persoana R. M.
A fost creata persoana R. M., Manager de Departament
A fost creat angajatul cu ID 7
A fost creat un manager, R. M., Manager de Departament
A fost creata structura departamentului Financiar
A fost creata organizatia
```

OC.3. Crearea claselor Java

OC.3.1. Structura codului unei clase Java

Structura codului unei clase Java este urmatoarea:

```
declaratie clasa {
  implementare (corp) clasa
}
```

sau, **detaliind elementele corpului clasei**:

```
declaratie clasa {
  declaratii atribute (variabile membru)
  declaratii constructori (functii de initializare a obiectelor)
  declaratii metode (functii membru)
}
```

Exemplu de clasa Java:

```
1   import java.util.Vector;
2   import java.util.EmptyStackException;
3
4   public class Stack              // declaratia clasei
5   {                               // inceputul corpului clasei
6
7     private Vector elemente;                // atribut (variabila membru)
8
9     public Stack() {                        // constructor
10      elemente = new Vector(10);            // (functie de initializare)
11    }
12
13    public Object push(Object element) {    // metoda
14      elemente.addElement(element);         // (functie membru)
15      return element;
16    }
17
18    public synchronized Object pop(){       // metoda
19      int lungime = elemente.size();        // (functie membru)
20      Object element = null;
21      if (lungime == 0)
22        throw new EmptyStackException();
23      element = elemente.elementAt(lungime - 1);
24      elemente.removeElementAt(lungime - 1);
25      return element;
26    }
27
28    public boolean isEmpty(){               // metoda
29      if (elemente.size() == 0)             // (functie membru)
30        return true;
31      else
32        return false;
33    }
34
35  }                               // sfarsitul corpului clasei
```

OC.3.2. Elementele declarației de clasa în Java

Formatul general al declaratiei unei clase Java este:

```
[public] [abstract] [final] class NumeClasa [extends NumeSuperclasa]
                                [implements  NumeInterfata [, NumeInterfata]] {
  // Corp clasa
}
```

Elementele opționale se afla intre paranteze drepte, **identificatorii** sunt scrisi cu text inclinat (*italic*), iar **cuvintele cheie** cu text drept.

Prin convenție, **numele de clase Java încep cu literă mare**.

Declaratia minimala a unei clase Java (fara elemente optionale) este:

```
class NumeClasa {
  // Corp clasa
}
```

Dacă elementele opționale nu sunt declarate compilatorul Java presupune **implicit** despre clasa curent declarata ca:

- doar clasele din acelasi director (pachet) cu clasa curenta au acces la membrii clasei curente (**prietenie de pachet**),
- este **instantiabila** (se pot crea obiecte avand ca tip clasa curenta),
- **poate avea subclase** (create extinzand clasa curenta),
- **extinde clasa Object** (radacina ierarhiei de clase Java) si **nu implementeaza nici o interfata**.

In tabelul urmator sunt descrise **elementele declaratiei de clasa Java**.

Element al declaratiei clasei	Semnificatie
`public`	Orice cod exterior are acces la membrii clasei
`abstract`	Clasa nu poate fi instantiata (din ea nu pot fi create direct obiecte, ci doar din subclasele ei non-abstracte)
`final`	Clasa nu poate avea subclase
`class` ***`NumeClasa`***	Numele clasei este ***`NumeClasa`***
`extends` ***`NumeSuperClasa`***	Clasa extinde o superclasa ***`NumeSuperClasa`*** (este o subclasa a clasei ***`NumeSuperClasa`***)
`implements` ***`NumeInterfata`***	Clasa implementează o interfata ***`NumeInterfata`***
`{ // Corp clasa }`	

Se observa ca **o clasa nu poate fi declarata in acelasi timp** si `abstract` si `final`, deoarece o clasa declarata `abstract` trebuie extinsa pentru a avea subclase concrete (tipuri din care pot fi create obiecte) pe cand unei clase declarate `final` ii este interzis sa fie extinsa.

OC.3.3. Corpul clasei în Java

Corpul clasei în Java conține:

- **constructori**, functii pentru ***inițializarea obiectelor***,
- **declarații ale variabilelor** care ***compun starea*** clasei şi obiectelor,
- **metode** care ***implementează comportamentul*** clasei şi obiectelor.

Variabilele şi metodele Java sunt denumite împreună **membri** Java. Constructorii nu sunt membri.

OC.3.4. Atributele (variabilele membru) Java

OC.3.4.1. Declararea atributelor (variabilelor membru) Java

Formatul general al declaratiei unui atribut (variabile membru) Java este:

```
[nivelAcces] [static] [final] [transient] [volatile] tipAtribut numeAtribut;
```

unde ***nivelAcces*** poate fi **`public`**, **`protected`** sau **`private`**. Prin convenție, **numele de variabile Java (inclusiv atributele) încep cu literă mică**.

Declaratia minimala a unui atribut Java (fara elemente optionale) este:

```
tipAtribut numeAtribut;
```

Dacă elementele opționale nu sunt declarate compilatorul Java presupune **implicit** ca:

- **doar clasele din acelasi director cu clasa curenta au acces la atributul curent**,
- **atributul are caracter de obiect** (fiecare obiect din clasa curenta are un astfel de atribut nepartajat cu alte obiecte, creat dinamic in momentul crearii obiectului),
- **valoarea atributul poate fi modificata oricand** (este o variabila).

In tabelul urmator sunt descrise **elementele declaratiei unui atribut Java**.

Element al declaratiei atributului	Semnificatie
`public`	Orice cod exterior clasei are acces la atribut
`protected`	Doar codul exterior din sublcase sau aflat in acelasi director are acces la atribut
`private`	Nici un cod exterior nu are acces la atribut
`static`	Are **caracter global**, de clasa (este o variabila creata static, odata cu clasa, a carei **locatie unica** este **partajata de toate obiectele clasei**)
`final`	Valoarea atributului nu poate fi modificata dupa initializare (este o **constanta**)
`transient`	Semnificatia nu este complet specificata (dar tine de serializarea obiectelor)
`volatile`	Previne compilatorul de la efectua anumite optimizari asupra atributului
`tipAtribut numeAtribut`	Tipul este ***`tipAtribut`*** iar numele este ***`numeAtribut`***
`[ = valoareInitiala];`	Eventuala initializare

In aceeasi clasa **nu pot fi declarate mai multe atribute cu acelaşi nume**.

Un **atribut dintr-o subclasă ascunde** un **atribut cu acelaşi nume** (si de acelasi tip) **din superclasă**.

În plus, **o variabilă membru şi o metodă membru pot avea acelaşi nume**. De exemplu, următorul cod Java este legal:

```
public class Stack  {
  private Vector elemente;        // atribut (variabila membru)

  public Vector elemente() {      // metoda (functie membru) cu acelasi nume
    return elemente;
  }

  // ...
}
```

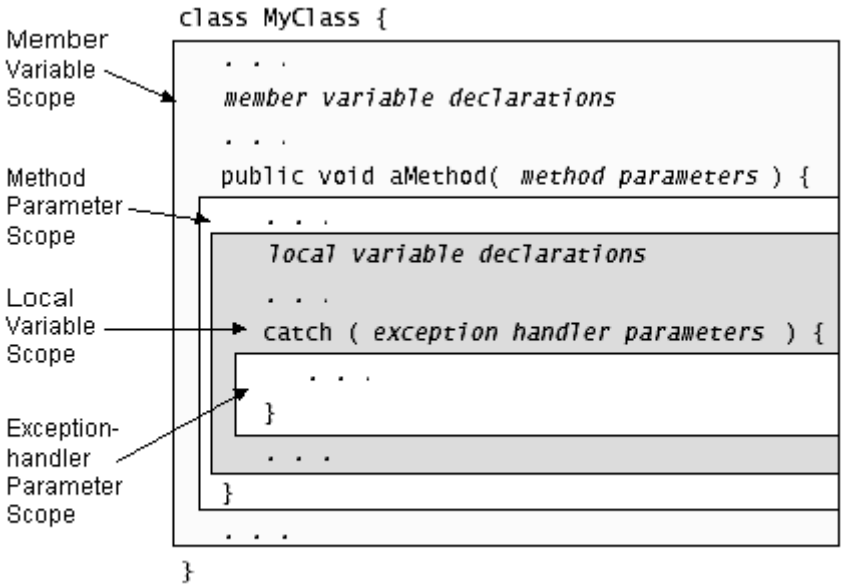
OC.3.4.2. Scopul variabilelor Java

Scopul variabilelor Java (vizibilitatea lor in interiorul clasei):

- reprezintă **portiunea de cod al clasei în care variabila este accessibilă** si
- **determină momentul în care variabila este creată şi distrusă**.

Exista 4 categorii de scop al variabilelor Java:

- **variabilă membru** (*member variable*) sau **atribut**,
- **variabilă locală** (*local variable*),
- **parametru al unei metode** (*method parameter*),
- **parametru al unei proceduri de control al exceptiilor** (e*xception-handler parameter*),



Variabila membru (atributul):

- este membrul unei clase sau al unui obiect,
- poate fi declarată oriunde în clasă, dar nu într-o metodă,
- e **disponibilă în tot codul clasei**.

Variabila locală:

- poate fi declarată oriunde într-o metodă sau într-un bloc de cod al unei metode,
- e **disponibilă în codul metodei, din locul de declarare şi până la sfârşitul codului metodei, sau până la sfârşitul blocului de cod al metodei**.

Parametrul unei metode:

- este argumentul formal al metodei,
- este utilizat pentru a se pasa valori metodei,
- e **disponibil în întreg codul metodei**.

Parametrul unui *handler* de excepție:

- este argumentul formal al *handler-ului* de excepție,
- este utilizat pentru a se pasa valori *handler-ului* de excepție,
- e **disponibil în întreg codul *handler-ului* de excepție**.

Urmatorul cod Java ilustreaza diferentele intre **atribute**, **variabile locale** si **parametri ai unor metode**.

```
 1    public class Complex        // declaratia clasei
 2    {                           // inceputul corpului clasei
 3
 4      private double real;            // real = atribut
 5      private double imag;            // imag = atribut
 6
 7      public void setReal(double real) {    // metoda, real = parametru
 8
 9        this.real = real;                   // real = atributul, real = parametrul
10      }
11
12      public void.setImag(double imag) {    // metoda, imag = parametru
13
14        this.imag = imag;                   // imag = atributul, imag = parametrul
15      }
16
17      public static void main(String[] args) {      // metoda, args = parametru
18
19        double real = Double.parseDouble(args[0]);  // real = variabila locala
20        double imag = Double.parseDouble(args[1]);  // imag = variabila locala
21
22        Complex c = new Complex();                  // c = variabila locala
23
24        c.setReal(real);                            // c, real = variabilele locale
25        c.setImag(imag);                            // c, imag = variabilele locale
26
27        System.out.println("{" + c.real +           // c.real = atributul lui c
28                           ", " + c.imag + "}");    // c.imag = atributul lui c
29      }
30    }                           // sfarsitul corpului clasei
```

OC.3.5. Constructorii (initializatorii) obiectelor Java

Constructorul Java este un **tip special de funcție Java**:

- este **utilizata pentru a inițializa un nou obiect de acel tip** (la Java în momentul creării dinamice a obiectului);
- **are acelaşi nume cu numele clasei** a cărei membră este;
- **nu returnează** nici o valoare;
- are aceleaşi **grade de accesibilitate**, **reguli de implementare a corpului** şi **reguli de supraîncărcare a numelui** cu funcțiile membre/metodele obişnuite.

Formatul general al declaratiei unui constructor Java este:

```
[nivelAcces] NumeClasa( listaParametri ) {
  // Corp constructor
}
```

unde ***nivelAcces*** poate fi **public**, **protected** sau **private**.

Declaratia minimala a unui constructor Java (fara elemente optionale) este:

```
NumeClasa() {
  // Corp constructor
}
```

Dacă elementele opționale nu sunt declarate compilatorul Java presupune **implicit** despre constructorul curent declarat ca doar clasele din acelasi director cu clasa curenta au acces la el.

In tabelul urmator sunt descrise **elementele declaratiei unui constructor Java**.

Element al declaratiei constructorului	Semnificatie
public	Orice cod exterior clasei are acces la constructor
protected	Doar codul exterior din sublcase sau aflat in acelasi director are acces la constructor
private	Nici un cod exterior nu are acces la constructor
NumeClasa	Numele constructorului este ***NumeClasa***
(*listaParametri*)	Lista de parametri primiti de constructor, despartiti prin virgule, cu formatul ***tipParametru numeParametru***

În Java nu este neapărat necesară scrierea unor constructori pentru clase. Un **constructor implicit** este **generat automat de sistemul de execuție pentru orice clasă care nu conține constructori**. Acest constructor nu face nimic (nici o inițializare). De aceea, orice initializare dorită impune scrierea unor constructori.

Java suportă **supraîncărcarea numelor** pentru constructori astfel încât o clasă poate avea orice număr de constructori, toți având acelaşi nume, dar **liste de parametri diferite**.

```
public Stack() {
    elemente = new Vector(10);
}

public Stack(int lungimeInitiala) {
    elemente = new Vector(lungimeInitiala);
}
```

Tipic, **un constructor utilizează argumentele sale pentru a initializa starea noului obiect**. În momentul creării unui obiect, compilatorul alege constructorul ale cărui argumente se potrivesc modului de inițializare utilizat de programator pentru a initializa noul obiect.

Compilatorul îi diferențiază pe constructori pe baza numărului de parametri din listă şi a tipului lor. Compilatorul ştie că atunci când găseşte codul următor, el trebuie să utilizeze constructorul care cere un singur argument întreg (utilizat pentru stabilirea lungimii maxime a stivei):

```
new Stack(10);
```

De asemenea, scriind codul următor, compilatorul alege constructorul implicit (fără argumente):

```
new Stack();
```

OC.3.6. Metodele (functiile membru) Java

OC.3.6.1. Declaratia metodelor Java

Formatul general al declaratiei unei metode (functii membru) Java este:

```
[nivelAcces] [static] [abstract] [final] [native] [synchronized] tipReturnat
          numeMetoda ( [listaDeParametri] ) [throws NumeExceptie [,NumeExceptie] ]
{
  // Corp metoda
}
```

unde ***nivelAcces*** poate fi **`public`**, **`protected`** sau **`private`**. **Prin convenție, numele de metode Java încep cu literă mică.**

Declaratia minimala a unei metode Java (fara elemente optionale) este:

```
tipReturnat numeMetoda() {
  // Corp metoda
}
```

Dacă elementele opționale nu sunt declarate compilatorul Java presupune **implicit** ca:

- **doar codurile claselor din acelasi director cu clasa curenta au acces la metoda curenta**,
- **metoda are caracter de obiect** (este creata dinamic in momentul crearii obiectului),
- **metoda este implementata** (are corp),
- **metoda poate fi rescrisa (reimplementata) in subclase** (create extinzand clasa curenta),
- **metoda este implementata in Java**
- **metoda nu are protectie la accesul concurent la informatii partajate**
- **metoda nu are parametri,**
- **metoda nu "arunca" (declanseaza) exceptii**.

In tabelul urmator sunt descrise **elementele declaratiei unei metode Java**.

Element al declaratiei metodei	Semnificatie
`public`	Orice cod exterior clasei are acces la metoda
`protected`	Doar codul exterior din sublcase sau aflat in acelasi director are acces la metoda
`private`	Nici un cod exterior nu are acces la metoda
`static`	Are caracter global, de clasa (este creata static, odata cu clasa)
`abstract`	Nu are implementare (trebuie implementata in subclase) si impune declararea `abstract` a clasei din care face parte (prin urmare clasa din care face parte nu poate avea instante)
`final`	Nu poate fi rescrisa implementarea metodei
`native`	Metoda implementata in alt limbaj
`synchronized`	Are protectie la accesul concurent la informatii partajate
`tipReturnat numeMetoda`	Tipul returnat este ***`tipReturnat`*** iar numele ***`numeMetoda`***
`(` ***`listaParametri`*** **`)`**	Lista de parametri primiti de metoda, despartiti prin virgule, cu formatul ***`tipParametru numeParametru`***
`throws` ***`NumeExceptie;`***	Metoda arunca exceptia ***`NumeExceptie`***

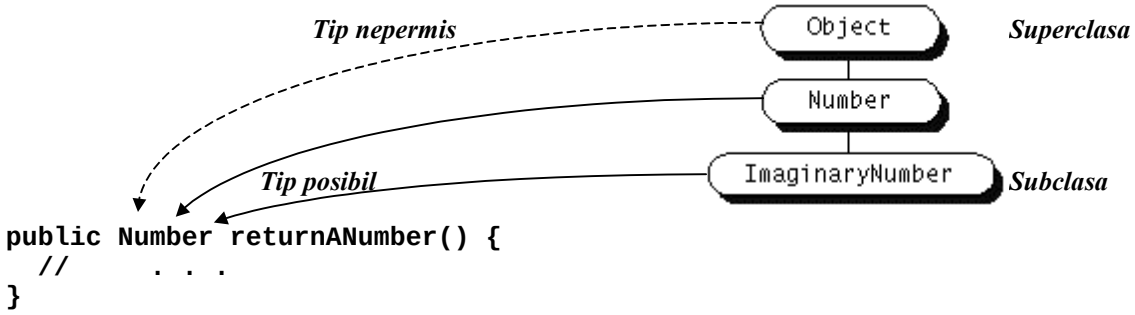
OC.3.6.2. Valoarea returnată de o metodă Java

Metodele Java pot returna:

- tip primitiv (int, char, boolean, real, etc.);
- tip referință (obiect, tablou, interfață).

Când metoda Java returnează un obiect, **clasa obiectului returnat trebuie să fie**:

- fie ***o subclasă***,
- fie ***chiar clasa*** tipului returnat.



OC.3.6.3. Numele metodelor Java

Java suportă **supraîncărcarea numelor** (***name overloading***) **funcțiilor membru/metodelor** astfel încât mai multe metode po partaja acelaşi nume, prin utilizarea unor liste de parametri diferite (prin numărul / ordinea / tipul parametrilor).

O clasă Java poate utiliza **rescrierea** (***overriding***) **unei funcții membru/metode** a superclasei sale.

Funcția membru/metoda rescrisă trebuie să aibă exact acelaşi nume, tip returnat şi listă de parametri ca metoda pe care o rescrie.

OC.3.6.4. Pasarea informatiilor (parametrilor, argumentelor) unei metode

Forma generală a declarațiilor din lista de parametri formali (care poate fi nulă, dar şi formată din mai mulți parametri - declarațiile lor apărând despărțite prin virgulă):

```
type name
```

Tipurile argumentelor Java pot fi:

- tip **primitiv** (int, char, boolean, real, etc.);
- tip **referință** (la obiect, tablou, interfață),

dar **nu pot fi metode**.

Numele argumentelor Java sunt utilizate în corpul metodei pentru referirea la informația pasată.

Un argument al unei metode poate avea acelaşi nume cu una dintre variabilele membru ale clasei. În acest caz, se spune că **argumentul ascunde** (***hides***) **variabila membru**.

Exemplu Java:

```
class Circle {
  int x, y, radius;
  public Circle(int x, int y, int radius) {
       // . . .
  }
}
```

`x, y` sau `radius` în corpul constructorului **referă argumentul** cu numele respectiv, **nu referă variabila membru**.

Pentru accesul la variabila membru, ea **trebuie referită prin** intermediul `this`.

```
class Circle {
  int x, y, radius;
  public Circle(int x, int y, int radius) {
    this.x = x;
    this.y = y;
    this.radius = radius;
  }
}
```

Pasarea informațiilor în Java este totdeauna prin valoare, ceea ce înseamnă că **argumentele de tip primitiv sau referință nu pot fi modificate**, dar **valorile "interne" ale tipurilor referință** (elementele tablourilor şi membrii obiectelor) **pot fi modificate**.

Astfel, **pot fi invocate metodele şi modifica variabilele accesibile ale obiectelor ale căror referințe sunt pasate**.

OC.3.6.5. Corpul metodei Java

Pe lângă elementele obişnuite de limbaj Java, pot fi utilizate în corpul metodei cuvinte cheie:

- pointerul `this` - pentru a referi obiectul curent;
- `super` - referirea/apelul membrilor superclasei.

Numele argumentelor au precedență față de cele ale variabilelor membru cu acelaşi nume şi le ascund pe acestea din urmă. Pointerul `this` este utilizat în acest caz **pentru a deosebi membrii obiectelor curente de argumente**.

Pointerul `this` mai poate fi utilizat, redundant, **pentru a preciza mai clar clasa obiectului**.

Se poate utiliza `super`, dacă metoda curentă **ascunde** (***hides***) o variabilă membru a superclasei, **pentru a referi variabila ascunsă**.

De asemenea, se poate utiliza `super`, dacă metoda curentă **rescrie** (***overrides***) o metodă membru a superclasei, **pentru a invoca metoda rescrisă**.

OC.3.7. Controlul accesului la membri Java - nivelurile de încapsulare

Clasele permit protecția variabilelor şi metodelor membru de accesul altor obiecte.

Regulile de vizibilitate completează / precizează noțiunea de **încapsulare**. Astfel, este posibilă obținerea unui **grad de încapsulare mai suplu**, dar şi protejarea, în beneficiul anumitor clase utilizator particulare, desemnate în specificația clasei furnizor.

Un scop al încălcării încapsulării poate fi **reducerea timpului de acces** şi a atributelor, obținută prin eliminarea necesității de a recurge la operații de selecție.

Efectul aplicării specificatorilor/modificatorilor Java:

acces specificator	cod clasă	pachet Java	cod subclasă	oricine	Observații
`public`	**da**	**da**	**da**	**da**	"***nici un secret***"
`protected`	**da**	**da**	**da**		"***secrete de familie (şi grup)***"
(`package` în Java)	**da**	**da**			"***secrete de grup***"
`private`	**da**				"***secrete absolute***" (dar obiectele din aceeaşi clasă au acces)

OC.3.8. Diferențele între membri de instanță şi membri de clasă în Java

Variabilele de instanță (declarate **fără** `static`) în Java:

- sunt **alocate şi proprii fiecărui obiect**.

Variabilele de clasă (declarate **cu `static`**) în Java:

- **alocate la nivel de clasă**, sunt **locații unice, partajate de toate obiectele clasei şi de clasă**,
- pot fi **referite atât cu numele instanțelor (obiectelor) cât şi cu numele clasei**.

Metodele de instanță (declarate **fără** `static`) în Java:

- au **acces atât la variabilele obiectelor cât şi la variabilele claselor**.

Metodele de clasă (declarate **cu `static`**) în Java:

- **nu pot accesa variabilele de instanță ale obiectelor** (decât dacă crează ele obiectele respective),
- pot fi **invocate atât cu numele instanțelor (obiectelor) cât şi cu numele clasei**.