

3. CLASE SI OBIECTE. TIPURI DE DATE ENUMERATE

3.1. Notiuni generale de programare obiectuala.....	<u>2</u>
3.1.1. Neajunsuri ale abordarii procedurale.....	<u>2</u>
3.1.2. Clasa: definitia unui nou tip de date compus.....	<u>2</u>
3.1.3. Ce avantaje are abordarea obiectuala?.....	<u>3</u>
3.1.3.1. Refolosirea codului.....	<u>3</u>
3.1.3.2. Incapsulare.....	<u>3</u>
3.2. Un scurt exemplu.....	<u>3</u>
3.3. Structura unei clase Java.....	<u>4</u>
3.3.1. Elemente de structura.....	<u>4</u>
3.3.2. Atributele (campurile) clasei.....	<u>5</u>
3.3.2.1. Particularitati fata de o variabila obisnuita.....	<u>5</u>
3.3.2.2. Definitia unui atribut.....	<u>5</u>
3.3.2.3. Accesarea atributelor obiectelor.....	<u>6</u>
3.3.3. Metodele clasei.....	<u>7</u>
3.3.3.1. Caracteristici generale.....	<u>7</u>
3.3.3.2. Definitia unei metode.....	<u>7</u>
3.3.3.3. Apelarea unei metode.....	<u>8</u>
3.3.3.4. Returnarea unei valori dintr-o metoda.....	<u>10</u>
3.3.3.5. Supraincarcarea metodelor.....	<u>10</u>
3.3.3.6. Metode cu numar variabil de argumente.....	<u>12</u>
3.4. Încapsularea.....	<u>12</u>
3.4.1. Necesitatea incapsularii si avantajele sale.....	<u>12</u>
3.4.2. Modificatori si niveluri de acces.....	<u>13</u>
3.4.3. Metode de tip getter si setter.....	<u>14</u>
3.5. Constructorul.....	<u>15</u>
3.6. Distrugerea obiectelor.....	<u>17</u>
3.7. Relatia obiecte virtuale - obiecte reale.....	<u>17</u>
3.8. BIBLIOGRAFIE.....	<u>18</u>

3.1. Notiuni generale de programare obiectuala

3.1.1. Neajunsuri ale abordarii procedurale

Informatiile reale cu care doreste sa opereze programatorul sunt in general mai complexe decat primitivele cunoscute de catre masina virtuala Java, si in plus structura lor interna difera de la o aplicatie la alta. Spre exemplu, o aplicatie de gestionare de stocuri ar putea necesita un tip de date StocProdus, care sa includa denumirea produsului, cantitatea, data achizitiei etc. O alta aplicatie - sau chiar aceeasi - va dori poate un tip de date Produs, care sa includa diferitele caracteristici pe care orice produs gestionat de aplicatie le are. Desigur ca programatorul poate incerca sa rezolve problema folosindu-se de primitive, insa solutia va fi una complicata, nefiereasca si care nu ofera garantii: daca avem nevoie de o lista de stocuri, am putea crea cateva tablouri paralele - unul care memoreaza toate numele de produse, altul care stocheaza cantitatatile, inca unul pentru data achizitiei etc (desi nici chiar data achizitiei nu poate fi memorata direct intr-o singura variabila de tip primitiva!). Pentru a afla detaliiile celui de-al treilea stoc din lista am folosi indexul 2 in fiecare dintre tablouri pentru a extrage pe rand informatiile ce compun stocul. Aceasta solutie nu creeaza un tip de date, ci emuleaza existenta sa intr-un mod artificial, inefficient si incomod pentru programator.

Solutia eleganta este definirea unui tip de date distinct StocProdus, perceptuit ca atare de catre compilator si masina virtuala; stabilind componzitia sa de la bun inceput, se vor putea declara si folosi variabile de acel tip de date, iar compilatorul va putea verifica corecta lor utilizare in expresii.

Astfel de tipuri de date compuse apar prin alaturarea de variabile cu tipuri de date deja existente; spre exemplu, pentru a reprezenta o data calendaristica, ar fi suficient sa cream un tip de date care inglobeaza 3 variabile de tip int - pentru zi, luna si an. In schimb, pentru tipul de date StocProdus, avem nevoie de o variabila String pentru numele de produs, una de tip int pentru cantitate si una de tip data calendaristica pentru data achizitiei.

Lucrurile nu se opresc insa aici; presupunand ca limbajul ne pune la dispozitie o modalitate de a ne crea propriile tipuri de date, chiar daca fiecare data calendaristica va avea componzitia corecta, nimeni nu ne poate garanta validitatea informatiilor componente; ziua, luna si anul fiind de tip int, ele accepta si valori negative sau mai mari decat cele admise (ziua >31, luna >12)! Considerand o variabila de tip data calendaristica, este necesara impunerea unor restrictii pentru fiecare dintre cele 3 elemente componente, astfel incat valorile lor sa aiba intotdeauna sens. Acest lucru se realizeaza - dupa cum se va vedea - „ascunzand” elementele componente ale datei si fortand orice actiune aplicata lor sa treaca printr-un set de functii atasate care asigura permanent validitatea informatiei.

3.1.2. Clasa: definitia unui nou tip de date compus

O clasa reprezinta structura unui nou tip de date definit de catre programator, spre deosebire de primitive a caror structura este deja cunoscuta compilatorului si masinii virtuale. Pe baza clasei putem construi **obiecte**, care sunt date de acel nou tip; crearea de obiecte ale unei clase poarta numele de **instantiere** a clasei in cauza. Fiecare obiect este o **instanta** (incarnare) a clasei, asa cum mai multe aparate de acelasi fel pot fi construite pornind de la aceeasi schema electronica. La clase diferite vor corespunde tipuri de obiecte diferite, asa cum din doua scheme electronice distincte obtinem doua tipuri de aparate distincte.

O clasa este un şablon pentru obiecte – ea specifica compozitia si modul de manifestare al obiectelor ce se vor crea pe baza ei. Analogii:

- o clasa este ca schema electronica pentru un aparat: schema specifica componentele, modul de așezare si felul in care interacționează ele; pe baza ei pot fi construite nenumărate aparate
- o clasa este ca o rețetă culinara: specifica un set de ingrediente si combinarea lor, iar pe baza ei se poate găsi de oricâte ori mâncarea in cauza

Un program Java devine astfel un ansamblu de obiecte care interacționează (își pot transmite mesaje unul altuia). Fiecare dintre ele acoperă o parte din scopul programului, astfel încât prin “colaborarea” intre obiecte se realizează funcția dorita. E ca si cum am lua un program scris intr-un limbaj procedural si l-am da mai multor programatori sa-l scrie pe porțiuni: fiecare va realiza o parte din el, si sub-programele vor trebui sa lucreze împreună in final. De remarcat insa ca lucrul cu

obiecte nu presupune simpla reunire a unor porțiuni de cod individuale, ci o abordare diferita a problemei fata de limbajele de programare procedurale: fiecare obiect este o entitate de sine statatoare care realizeaza o funcție bine stabilita, indiferent de cine îl va folosi pe viitor.

3.1.3. Ce avantaje are abordarea obiectuala?

3.1.3.1. Refolosirea codului

Limbajele orientate pe obiect încearcă sa crească eficiența programării printr-o mai buna refolosire a codului și modularizare. Un nou tip de date, odată ce a fost definit, testat si este funcțional, poate fi folosit atât de către alte obiecte din același program cat si de către alți programatori in aplicațiile lor, deoarece formează o entitate de sine statatoare, reutilizabilă. Sa ne gândim la o clasa care implementează o agenda de telefon: aceasta va conține datele efective (numerele) si modalități de adaugare/stergere/editare a acestora, cu verificările de rigoare. Un obiect de acest tip poate fi folosit in alt program (sa zicem, un soft de trimis faxuri) fără sa fie necesara scrierea de cod pentru partea de management al numerelor de telefon, ci doar cunoscând modul de "manevrare" a unui obiect de tip agenda (felul in care cautam/citim/salvam un număr etc.). De fapt, avem de a face cu o modelare a realitatii, unde, daca avem nevoie de un obiect/aparat/etc care realizeaza o anumita functie, ni-l cumparam si devenim astfel simpli utilizatori ai obiectului, fara a avea neaparat cunostinte in privinta structurii sale interne.

Odata cu programarea obiectuala au fost introduse si noi modalitati de refolosire de cod. Dupa cum se va vedea, unul dintre conceptele fundamentale in programare orientata pe obiect - mostenirea - are ca avantaj si o foarte eficienta refolosire a codului.

3.1.3.2. Incapsulare

Nu știm din ce este format si cum funcționează intern un aparat de radio, dar știm cum să-l utilizam. De asemenea, chiar daca am vrea, nu putem interveni direct in semnalele electrice care circula in interiorul aparatului, deoarece acesta este închis intr-o carcasa opaca; singurele lucruri de care dispunem sunt butoanele si afişajul. Nu putem schimba frecventa de lucru, de exemplu, decât acționând un potențiometru; intern, are loc o modificare o parametrilor unui filtru care are ca rezultat final acordarea aparatului pe alta frecventa. Acțiunile noastre externe au corespondent in modificarea starii interne a aparatului, insa nu putem opera direct asupra componentelor electronice ce memoreaza aceasta stare interna.

In același fel, un obiect **încapsulează** datele cu care lucrează (cele care compun starea sa interna) lăsând la dispoziția utilizatorului numai **interfața** ce permite manevrarea - indirecta! - a datelor încapsulate. Incapsularea presupune ascunderea datelor in interiorul obiectului si fortarea accesului la ele prin functii special scrise, care asigura validitatea permanenta a acestor date. Utilizatorul obiectului modifica starea sa interna indirect, prin intermediul acestor functii, fara sa aiba acces la miezul sensibil al obiectului.

Acest mod de abordare ofera avantaje multiple:

- mentinerea validitatii datelor - accesul la datele stocate in obiect nu se face direct, ci prin intermediul unor functii intermediare
- decuplare - cat timp utilizatorul unui obiect apeleaza aceleasi functii si in acelasi mod, programatorul poate schimba implementarea interna a obiectului (poate remedia bug-uri sau aduce optimizari). Prin analogie, in viata reala, putem cumpara o noua versiune a unui aparat care, intern, este radical modificat dar pe care, in virtutea faptului ca dispune de aceleasi butoane pe carcasa si care indeplinesc aceleasi functii, il vom putea in continuare utiliza la fel ca pe cel vechi
- dezvoltare si mentenanta mai usoara - crearea de clase si obiecte duce automat la modularizarea codului si deci la dezvoltarea/depanarea sa structurata, pe portiuni

3.2. Un scurt exemplu

Fie definitia unui nou tip de date numit Persoana:

```
class Persoana{
    String nume;
    int greutate = 50;

    public void prezenta(){
        System.out.println("Salut, sunt " + nume + " si am " + greutate + " de kg");
    }

    public void schimbaNume(String n){
        nume = n;
    }

    public void mananca(){
        greutate++;
    }
}
```

Fiecare obiect Persoana va avea propriile caracteristici *nume* și *greutate*, și asupra sa vor putea fi aplicate trei acțiuni: persoanei i se poate cere să se hranească sau să se prezinte și i se poate seta/schimba numele. Aceste acțiuni sunt echivalentul butoanelor de pe carcasa în cazul aparatelor reale.

Odată tipul de date definit, se pot declara variabile de acest nou tip de date:

```
Persoana p;
```

Atenție! Variabila *p* este automat una de tip referință!

Pe moment nu există niciun obiect Persoana, ci doar definitia acestui tip de date și o variabilă de noul tip. Pentru a crea obiecte este folosit operatorul **new**, familiar cititorului de la capitolul dedicat tablourilor și care, ca și acolo, este responsabil cu alocarea de memorie:

```
p = new Persoana();
```

Odată obiectul creat, ii putem “apăsa butoanele”, după cum urmează:

```
p.schimbaNume("Mihai");
p.prezenta();           // Salut, sunt Mihai și am 50 de kg
p.mananca();
p.prezenta();           // Salut, sunt Mihai și am 51 de kg
p.mananca();
p.prezenta();           // Salut, sunt Mihai și am 52 de kg
p.schimbaNume("Vlad");
p.prezenta();           // Salut, sunt Vlad și am 53 de kg
```

Se observă că obiectul își menține starea internă între două “apsari pe butoane”; fiecare nouă acțiune aplicată obiectului are ca punct de pornire starea în care l-a lăsat acțiunea anterioară.

3.3. Structura unei clase Java

3.3.1. Elemente de structură

O definitie de clasa conține atât informația efectiva ce va fi continuata în obiecte, cat și modalitățile de manevrare a acesteia. Din punct de vedere formal, definitia de clasa este formata din **membri ai clasei**, care sunt de două tipuri:

- **câmpuri (attribute)** – reprezintă datele conținute de către viitoarele obiecte (echivalente cu lista de piese electronice ce compun un aparat). În exemplul de mai sus, variabilele *nume* și *greutate* sunt câmpuri ale clasei Persoana

- **metode** – funcții definite în cadrul clasei, care au acces la atribut și care permit manevrarea sigură și eficientă a acestora. O parte dintre ele sunt accesibile oricărui utilizator al obiectului și formează interfață acestuia cu exteriorul (echivalentul butoanelor pe care apăsăm în cazul unui aparat), altele putând fi inaccesibile în exterior și tinând de refolosirea internă a codului în cadrul obiectului. În cazul clasei Persoana de mai sus, funcțiile *schimbaNume()*, *prezentare()* și *mananca()* reprezintă metode ale clasei.

După cum se observă din exemplul anterior, noțiunile introduse nu sunt complet noi: campurile sunt niste variabile, iar metodele sunt niste funcții, astăzi vor fi în continuare valabile multe dintre noțiunile pe care cititorul le are din limbaje procedurale. Se va vedea însă că există și deosebiri semnificative față de variabilele și funcțiile clasice - ele vor fi punctate pe măsură ce subiectul este dezvoltat.

3.3.2. Atributele (campurile) clasei

3.3.2.1. Particularități față de o variabilă obisnuită

Un camp al clasei reprezintă o variabilă declarată ca membru al clasei - în interiorul clasei, dar în afara metodelor acesteia. Există următoarele particularități față de o variabilă obisnuită:

- variabilă se va multiplica odată cu instantierea clasei. Fiecare obiect conține propria copie a setului de atribut din definirea clasei. Este și firesc, dacă ne gândim că fiecare persoană, spre exemplu, are propriul sau nume și propria greutate, distincte de cele ale altor persoane
- variabilă este vizibilă cu numele sau scurt în cadrul clasei, dar nu și în afara acesteia. Având în vedere că la un moment dat pot exista multe copii ale aceleiasi variabile (cate una per obiect), este necesară precizarea obiectului căruia variabilă este accesată
- definirea variabilei poate fi precedată de un asa-numit *modificator de acces* (public, private sau protected) - acesta este însă optional. În acest fel, campul devine accesibil utilizatorului obiectului sau ascuns în interiorul obiectului

Ansamblul valorilor atributelor (considerat la anumit moment) definește starea obiectului la acel moment. Toate obiectele unei clase au aceeași structură internă - ceea ce le deosebesc este faptul că valoarea pentru același camp poate差别 de la un obiect la altul. Trei obiecte Persoana pot avea nume diferenți sau/si greutăți diferențiale.

Nota: chiar și în condițiile în care două obiecte au aceleasi valori pentru toate atributele din definirea clasei, ele continuă să fie obiecte diferențiate - ocupă spații distincte de memorie și implicit sunt accesate cu referințe diferențiale!

Atributele sunt automat vizibile (accesibile) în metodele clasei, indiferent de modificatorul de acces atașat, însă nu neapărat și în afara acesteia. Atunci când modificatorul de acces permite vizibilitatea unui atribut în afara clasei, acesta oricum nu va putea fi accesat cu numele sau scurt (vezi capitolul despre accesarea atributelor).

3.3.2.2. Definiția unui atribut

Sintaxa definiției unui atribut de clasa este următoarea (portiunile cuprinse între <...> sunt opționale):

```
<modificator_de_acces> tip_de_date nume_variabila <= valoare> ;
```

Conform convențiilor de codare Java, numele campului trebuie să înceapă cu litera mică și, dacă este format din mai multe cuvinte, fiecare cuvânt ulterior trebuie să înceapă cu litera mare:

```
int greutate;
String serieNumarBuletin;
```

Tipul de date al campului poate fi primitivă sau clasa; în cazul al doilea, variabilă va fi de tip referință și va indica un alt obiect (în acest fel putem crea sensația că un obiect „conține” altul):

```
class Sofer{
    // ...cu o implementare oarecare...
}

class Autovehicul{
    int anInmatriculare;
    Sofer s; // un autovehicul "contine" un sofer (de fapt o referinta catre un obiect Sofer)
}
```

Definitia unui camp poate contine ca prin element un modificador de acces: public, private sau protected. Acestea vor fi tratati in cadrul acestui material intr-un capitol ulterior.

Un camp poate fi sau nu initializat la declarare:

- daca atributul nu este initializat la declarare, la crearea unui nou obiect el va primi automat valoarea default a tipului de date respectiv (0 pentru numere, false pentru boolean, null pentru referinte etc)
- daca atributul este initializat la declarare, la crearea unui nou obiect el va primi valoarea specificata.

In ambele cazuri, efectul este ca toate obiectele aceleiasi clase au imediat dupa creare aceeasi stare initiala.

3.3.2.3. Accesarea atributelor obiectelor

Un atribut nu este o variabila de sine statatoare, ci apartine de un anumit obiect. De aceea, campul trebuie accesat specificand si referinta catre obiectul de care apartine, dupa cum urmeaza:

```
referinta.numeAtribut
```

Referirea unui atribut se realizeaza diferit in functie de locul in care se afla codul care acceseaza atributul:

- daca se incercă accesarea atributului din interiorul clasei (din cadrul unei metode), se poate scrie `this.numeAtribut` sau simplu `numeAtribut`. Cuvantul cheie **this** este o referinta predefinita catre pointeaza intotdeauna catre obiectul curent
- daca se incercă accesarea atributului din afara clasei, aceasta inseamna ca programatorul care utilizeaza clasa (acelasi cu cel care a creat-o sau altul) a instantiat-o, dispune de o referinta catre un obiect al ei si doreste sa acceseze atributul *acelei* obiect

```
class Persoana{
    public int IQ;
    public void invata(){
        this.IQ++; // accesarea atributului din interiorul clasei
        // ar fi functionat si IQ++;
    }
}

class Main{
    public static void main(String[] args){
        Persoana p = new Persoana();
        p.IQ = 140; // accesarea atributului din afara clasei, prin intermediul unei
        // referinte la un obiect al clasei
    }
}
```

Cuvantul cheie **this** devine obligatoriu atunci cand avem cazuri de variable shadowing - o variabila locala care are acelasi nume cu un camp al clasei:

```
class C2 {
    int x=4;
    void setX(int x){
        this.x=x; // this.x refera campul x al clasei, iar simplul nume x refera
        // variabila locala (argumentul metodei). Asadar, in caz de suprapunere
        // de nume, variabilele locale au intaiatate
    }
}
```

3.3.3. Metodele clasei

3.3.3.1. Caracteristici generale

Metodele sunt functii definite in cadrul unei clase. Ele pastreaza in buna parte proprietatile functiilor pe care le cunoastem din limbajele procedurale:

- reprezinta un bloc de instructiuni ce primeste un nume si care poate fi apelat apoi in mod repetat prin intermediul acestui nume
- pot primi date de intrare si pot produce date de iesire. In acest fel, desi algoritmul (blocul de instructiuni) este acelasi, rezultatele de iesire produse pot diferi de la o executie la alta
- in cadrul unei metode se pot defini variabile, insa nu si alte metode

Metodele prezinta si diferente semnificative fata de functiile din limbajele procedurale:

- codul (instructiunile) este plasat aproape exclusiv in metode. In multe limbaje procedurale, crearea unei functii era o chestiune de alegere, pentru un program simplu instructiunile putand fi scrise direct in radacina fisierului sursa; in Java nu putem scrie instructiuni (afisare pe ecran, expresii etc) in afara unei definitii de clasa, iar in interiorul ei nu se poate scrie cod oriunde, ci doar in metode sau in putine alte constructii permise (ex: clase interioare, blocuri de initializare etc. - vezi capitolele urmatoare)
- domeniul lor de vizibilitate nu mai este global. In limbajele procedurale, o functie, odata definita, putea fi utilizata in orice punct (context) al programului. O metoda apartine unei clase, iar vizibilitatea sa in afara clasei este dictata de programator
- apelarea unei metode nu se mai realizeaza cu simplul ei nume (ex: *f()*) decat in cateva cazuri foarte particulare
- o metoda are acces neconditionat la campurile clasei din care face parte (indiferent de eventualul modificator de acces al acestora) insa accesul la alte variabile din program depinde de nivelul lor de acces si de relatiile intre clase

Proprietatea metodelor de a avea intotdeauna acces la atributele obiectului pe care opereaza le face sa actioneze ca intermediari in citirea si modificarea datelor din obiect. Ceea ce realizeaza de fapt limbajele obiectuale este o foarte stransa cuplare a datelor (campurile) cu codul care asigura manipularea lor (metodele): dupa cum se va vedea, putem obliga operatiile de citire si modificare a unui atribut sa treaca prin metode special definite care asigura permanent validitatea datelor.

Observatie: spre deosebire de cazul atributelor, codul din metodele unei clase este comun tuturor obiectelor instantiate pe baza clasei (nu se multiplică odata cu crearea de obiecte). Acest lucru este firesc deoarece toate obiectele de același tip se vor manifesta in același fel la „apasarea butonului” ce indeplinește aceeași acțiune.

3.3.3.2. Definitia unei metode

O metoda se definește astfel (portiunile incadrate intre <...> sunt optionale):

```
<modificatori> <calificatori> tip_date_produs nume (<tip_date_1 nume_arg1, tip_date2 arg2, ...>) {
    // instructiuni ce compun corpul metodei
}
```

Singurele elemente obligatorii in definitia unei metode sunt numele acesteia si tipul de date produs (numit si tip de date de intoarcere - „return type”).

Tipul de date al fiecarui argument in parte si cel de intoarcere (al datelor produse) pot fi oricare dintre tipurile de date deja existente, fie ele primitive, tablouri, clase predefinite sau clase definite anterior de catre programator, ca in exemplele urmatoare:

```
class ExempluMetode{
    public int adunare (int x, int y) {
        return x+y;
    }
}
```

```

public String numeComplet(String prenume, String numeFamilie){
    return prenume.concat(" ").concat(numeFamilie);
}

public StocProdus stocNou(String numeProdus, int cantitate, Date dataAchizitiei){
    return new SticProdus(numeProdus, cantitate, dataAchizitiei);
}

```

Faptul ca se specifica in definitia metodei toate aceste tipuri de date echivaleaza cu un angajament - unul pe care compilatorul se asigura ca ni-l vom respecta. In metoda *adunare()* din exemplu, ambele argumente au fost declarate *int*, si incercarea de a furniza un *float* sau un *String* la apelarea metodei va esua inca de la compilare. De asemenea, daca in cadrul metodei *numeComplet()* am fi uitat sa folosim instructiunea *return* sau am fi produs o valoare de alt tip de date decat *String*, compilarea ar fi generat o eroare atentionandu-ne ca nu ne-am indeplinit angajamentul luat in definitia metodei.

3.3.3.3. Apelarea unei metode

3.3.3.3.1. Sintaxa si particularitati

Apelarea unei metode are urmatoarea sintaxa generala:

```
referinta.numeMetoda(valoare_argument_1, valoare_argument_2,.....)
```

Valorile pasate ca argumente la apelarea unei metode pot fi fie valori ca atare („literals”), fie expresii care se vor evalua in momentul apelului. In ambele cazuri ele trebuie sa aiba tip de date identic sau compatibil cu cel al argumentului corespunzator din definitia metodei; compilatorul se va asigura de acest lucru, generand erori in caz de abatere. Spre exemplu, daca avem o metoda *f(int x)*, nu o vom putea apela cu un argument de tip *String* (ex: *f("test")*;) sau de tip primitiva la modul general, ci doar pasandu-i valori de tip *int*, *byte*, *short* sau *char*.

Pentru a justifica existenta referinte din sintaxa de apelare, sa ne amintim ca o metoda este o portiune de cod care, desi are aceeasi definitie pentru toate obiectele clasei, la apelare actioneaza asupra datelor unui *anume* obiect al clasei. Fie urmatoarea portiune de cod:

```

class Persoana{
    int greutate;

    public void mananca(){ greutate++; }

class Restaurant{
    public static void main(String[] args){
        Persoana p1 = new Persoana();
        Persoana p2 = new Persoana();
        p1.mananca();
        p2.mananca();
    }
}
mananca(); // eroare de compilare din cauza acestei linii!

```

Eroarea ce rezulta la compilare poate fi explicata din mai multe unghiuri:

- din cel al sintaxei: nu sunt permise instructiuni Java in afara definitiei unei clase
- din cel logic: chiar daca s-ar permite apelarea metodei *mananca()* in acel loc si cu acea sintaxa, compilatorul ar avea o dilema: a cui greutate trebuie incrementata, avand in vedere ca exista doua obiecte Persoana, fiecare cu greutatea lui?

Acesta este motivul pentru care o metoda este in general apelata pe baza unei referinte la obiectul vizat, in acelasi mod ca si campurile.

In functie de contextul codului in care este apelata metoda distingem doua cazuri:

- apelarea metodei din afara clasei in care a fost definita. Aceasta presupune ca intr-o alta clasa a fost creata o instanta a clasei noastre, catre care avem o referinta, si folosim acea referinta pentru a apela metoda (vezi apelurile catre metoda *mananca()* din clasa *Restaurant* de mai sus)
- apelarea metodei din interiorul clasei in care a fost definita - asadar din cadrul unei alte metode. Facem acest lucru atunci cand dorim sa refolosim cod. In acest caz metoda poate fi apelata cu numele sau scurt sau folosind referinta *this*:

```
class Persoana{
    void spalaMaini() { /* ...o implementare oarecare ... */ }
    void spalaDinti() { /* ...o implementare oarecare ... */ }
    void mananca() { greutate++; }
    void rutinaLuareMasa(){
        spalaMaini();
        mananca();
        spalaDinti(); // alternativ putem folosi this.spalaMaini(), this.mananca() etc
    }
}
```

Metodele clasei sunt intotdeauna vizibile una alteia, indiferent de eventualul lor modificador de acces.

Observatie: desi codul unei metode exista intr-un singur exemplar in memorie (indiferent de numarul de obiecte existente ale clasei), ea „stie” totusi pe atributele carui obiect sa actioneze! Acest lucru se datoreaza faptului ca o metoda este apelata prin intermediul unei referinte la un obiect; in interiorul metodei se foloseste (implicit sau explicit) *this*, initializat pe baza acestei referinte.

3.3.3.3.2. Pasarea argumentelor prin valoare

La apelarea unei metode Java, pasarea argumentelor se efectueaza prin valoare. Aceasta inseamna ca, daca folosim o variabila pe post de argument la apelul unei metode, va fi creata o copie a valorii acelei variabile, utilizata mai apoi in cadrul metodei:

```
class PasarePrinValoare{
    void f(int x){
        x++;
    }
    void g(){
        int a = 3;
        f(a);
        System.out.println(a); // 3 (desi poate ne-am fi asteptat la 4)
    }
}
```

In exemplul de mai sus, argumentul *x* al metodei *f()* este o variabila locala a acelei metode, initializata automat cu valoarea primita la apelare. Atunci cand se apeleaza *f(a)*, cand executia patrunde in metoda *f()* se va realiza practic atribuirea *x=a*, variabila locala *x* fiind initializata cu o copie a valorii lui *a*; din acest motiv, incrementarea ulterioara a lui *x* nu va afecta variabila *a*, deoarece modificarea este aplicata unei copii. De aceea, chiar daca obisnuim sa spunem („l-am pasat pe *a* ca argument in *f()*”), tehnic nu am pasat insasi variabila *a*, ci un duplicat al valorii ei.

Implicatia imediata a acestui mecanism este ca o variabila pasata ca argument la apelare nu poate fi modificata din cadrul metodei apelate: in exemplul de mai sus, chiar daca *a* a fost pasat ca argument, el nu poate fi modificat din metoda *f()*.

Concluziile de pana acum sunt valabile atat pentru variabile de tip primitiva cat si pentru referinte, insa in cazul referintelor exista o nuanta importanta. Fie exemplul urmator:

```

class PasareCuReferinte{
    void botez(Persoana x){
        x.nume="Mihai";
        x = null;
    }
    void test(){
        Persoana p = new Persoana();
        p.nume = "Ana";
        botez(p);
        System.out.println(p.nume); // Mihai (desi poate ne-am fi asteptat fie la Ana,
                                    // fie la o eroare din cauza referintei x ce ar fi null
    }
}

```

Ceea ce arata acest exemplu nu contrazice cele spuse pana acum, ci doar - eventual - intuitia noastra. La apelarea metodei *botez()* se efectueaza atribuirea *x=p*; datorita naturii variabilelor - referinte - *x* va pointa acum catre acelasi obiect ca si *p*! (caci, sa nu uitam, valoarea unei variabile de tip referinta reprezinta o modalitate de a ajunge la un obiect din memorie). Avem din acest moment doua referinte diferite, dar catre acelasi obiect. Indiferent prin intermediul careia dintre ele modificam obiectul, continutul acestuia VA FI afectat, si de aceea numele persoanei se schimba de data aceasta in Mihai. Pe de alta parte, modificand *referinta x* (nu obiectul catre care ea trimite!) din cadrul metodei *botez()*, nu il putem afecta pe *p*, ca dovada ca afisarea din final nu genereaza o eroare (daca *p* ar fi fost null, la incercarea de accesare a campului *p.nume* ar fi rezultat un *NullPointerException*).

3.3.3.4. Returnarea unei valori dintr-o metoda

Definitia unei metode include si tipul de date de intoarcere, care echivaleaza cu o promisiune de a produce date de acel tip sau de unul compatibil. Programatorul trebuie sa returneze o valoare folosind in cadrul metodei instructiunea *return* urmata de o valoare sau expresie. Odata „promisiunea” facuta, compilatorul se va asigura de doua lucruri:

- ca returnam o valoare daca am promis ca facem acest lucru. Compilatorul se va asigura ca exista cel putin o instructiune *return* in corpul metodei si ca aceasta se executa neconditionat. Daca compilatorul nu poate fi sigur ca returnam intotdeauna o valoare, codul nu va compila, ca in exemplul de mai jos:

```

class Returnare{
    int f(){
        if(Math.random()>0.9) {
            return 4;
        }
    }
/* eroarea de compilare este usor derutanta: missing return statement. Ce vrea sa spună
compilatorul este ca nu poate fi sigur ca în toate cazurile se va executa return. Ca dovada, dacă
adaugăm o ramură else care are propriul return, programul va compila corect */

```

- ca valoarea returnata are un tip de date identic sau compatibil cu cel promis. Daca tipul de date de intoarcere este *int*, nu vom putea returna *String*, *Persoana* sau *long*, in schimb vom putea produce o valoare de tip *int*, *byte*, *short* sau *char*!

Nota: atunci cand o metoda nu produce valori, programatorul indica acest lucru folosind void ca tip de date de intoarcere. In acest caz nu mai exista obligativitatea folosirii instructiunii return in corpul metodei. Daca totusi este folosita (ca simpla modalitate de control al executiei), ea nu trebuie urmata de o valoare!

3.3.3.5. Supraincarcarea metodelor

In cadrul aceleiasi clase Java este posibila definirea mai multor metode cu același nume – operațune numita **overloading (supraincarcare)**; compilatorul va distinge intre ele pe baza semnaturii acestora. **Semnatura** unei metode este ceea ce o individualizează la apelare: **numele sau si succesiunea tipurilor de date ale argumentelor**. Este ușor de distins intre doua metode care primesc același set de tipuri de argumente, daca au nume diferit; de asemenea, daca definim doua metode cu același nume, le putem deosebi după setul de parametri diferiți din punct de vedere al tipurilor de date, deoarece instructiunea de apelare va diferi.

```
class Numere {
    int x;
    // set de argumente diferit
    public int cresteNumar() { x++; }
    public int cresteNumar(int cuCat) { x+=cuCat; }
    public static void main(String[] args){
        Numere n = new Numere();
        n.cresteNumar();
        n.cresteNumar(4);    // apel cu argument; nu poate fi "incurcat" cu cel precedent
    }
}
```

In exemplul de mai sus, compilatorul trebuie sa decida, la fiecare apelare a unei metode *cresteNumar()*, ce varianta din definitia clasei sa utilizeze si deci ce implementare va include in bytecode corespunzator apelului in cauza. Hotararea este luata analizand succesiunea tipurilor de date ale argumentelor, asa cum sunt ele pasate la apelare: in primul caz nu avem argumente (si deci va fi folosita versiunea corespunzatoare de metoda *cresteNumar()*) iar in al doilea caz argumentul este de tip int, ceea ce duce catre cea de-a doua metoda *cresteNumar()*. Daca am fi apelat *cresteNumar("patru")*, compilatorul ar fi generat o eroare, deoarece nu exista nicio metoda *cresteNumar()* in definitia clasei care sa accepte ca unic argument un *String*. Pe de alta parte, *cresteNumar(b)* (unde b este de tip *byte*) ar fi functionat, fiind utilizata varianta *cresteNumar(int)*!

Mergand pe aceasta logica, concluzionam ca nu se poate face supraîncărcare folosind doar valori de întoarcere de tip diferit in definitiile celor doua metode - compilatorul nu va putea decide care dintre funcții trebuie apelata:

```
class c2 { // ASA NU!
    int x;
    public int intoarceX() { return x; }
    public long intoarceX() { return x+100000000000L; }
    public void f() { intoarceX(); } //pe care dintre functii o apelez??
}
```

De asemenea, supraîncărcarea nu se poate realiza prin simpla schimbare a numelor argumentelor:

```
class c3 { // ASA NU!
    int x;
    public int setX(int valoare) { x=valoare; }
    public int setX(int val) { x=val; }
}
```

deoarece compilatorul nu identifica (intern) o metoda după numele argumentelor, ci după succesiunea tipurilor de date ale acestora; in plus, in instructiunea de apelare a metodei nu intervin numele argumentelor!

Procedeul de supraincarcare este util pentru a da iluzia programatorului ca apeleaza aceiasi metoda dar in moduri diferite. In exemplul cu metoda *cresteNumar()*, perceptia utilizatorului clasei este ca metoda poate fi apelata fie fara argumente (producand incrementare), fie cu un argument care precizeaza cantitatea cu care trebuie marit numarul. Impresia finala este cea de flexibilitate. Trebuie inteles insa ca, atunci cand definim doua sau mai multe metode cu acelasi nume, pentru compilator si masina virtuala ele sunt metode complet distincte - de altfel, ele contin si seturi de instructiuni diferite.

Atunci cand avem metode supraincarcate exista deseori posibilitatea refolosirii de cod. O metoda poate apela o alta cu acelasi nume fara niciun fel de sintaxa speciala; iata o descriere a unui exemplu de mai sus:

```
class Numere2 {
    int x;

    public int cresteNumar(int cuCat) { x+=cuCat; }
    // varianta fara argument este un caz particular al implementarii anterioare:
    public int cresteNumar() { cresteNumar(1); }    // este apelata cealalta metoda cresteNumar
}
```

3.3.3.6. Metode cu numar variabil de argumente

Incepand cu Java 1.5 (denumita si Java 5), o metoda poate avea un numar variabil de parametri, cu conditia specificarii acestui fapt in definitia metodei. In acest scop exista argumente de tip special („varargs”) pe pozitia carora se pot specifica la apelare mai multe valori:

```
class VarArgs{
    static double medie(int...nr) {
        int suma = 0;
        for(int i=0;i<nr.length;i++) {
            suma += nr[i];
        }
        return ((double)suma)/nr.length;
    }

    public static void main(String[] args){
        System.out.println(medie(1,2,3));           //2
        System.out.println(medie(1,2,3,4,5));       //3
    }
}
```

Noul element de sintaxa este succesiunea de 3 puncte introdusa intre tipul de date si nume argumentului. In acest fel metoda din exemplu poate fi apelata cu ZERO sau mai multe argumente de tip int. Lista de valori in cauza este accesibila in interiorul metodei sub forma unui tablou de *int* avand numele argumentului (nr).

Nota: daca se paseaza zero argumente, tabloul va exista dar va fi gol (length=0)!

In definitia metodei pot exista atat argumente obisnuite cat si argumente cu numar variabil, cu urmatoarele conditii:

- poate exista un singur argument variabil
- acesta trebuie sa se afle la sfarsitul intregii liste de argumente, inainte de paranteza de inchidere

```
void friends(String nume, String...prieteni){
    System.out.println("Prietenii lui "+nume+" sunt:");
    for(String s:prieteni){
        System.out.println(s);
    }
}
```

Nota: metoda din exemplu trebuie apelata cu cel putin un argument de tip String, din cauza primului argument din definitie care nu este de tip multiplu.

3.4. Încapsularea

3.4.1. Necesitatea incapsularii si avantajele sale

Cu ceea ce avem pana acum putem crea clase in care se gasesc atribute si metode, insa ne lovim de o problema: cat timp atributele unui obiect sub disponibile direct utilizatorului sau, ele pot primi valori invalide. Fie urmatorul exemplu:

```
class Persoana{
    String nume;
    void modificaNume(String n){ nume = n; }
}

class CodClient{
    public static void main(String[] args){
        Persoana p = new Persoana();
        p.nume = "Ana";      // Ok
    }
}
```

```

        p.nume="uh&GJJ";      // Nu este ok, dar nu putem preintampina astfel de situatii
        p.nume = null;        // cat timp numele e accesibil direct prin intermediul lui p
    }
}

```

Pentru a asigura validitatea valorilor ce patrund in obiect se iau urmatoarele doua masuri:

- se interzice accesul direct la atributele sensibile, schimbandu-le acestora nivelul de acces folosind modificatori (vezi mai jos)
- se scriu metode speciale care intermediaza accesul la campuri, astfel incat orice incercare de modificare sau chiar de citire a valorii unui camp sensibil sa treaca printr-o sectiune de cod-filtru (vezi sectiunea despre metode getter si setter)

Exista doua implicații majore ale încapsulării:

- sunt lăsați "la vedere" numai acei membri care vor fi folosiți de catre utilizatorii obiectelor clasei; acești membri formează **interfața obiectului cu exteriorul** (cu celelalte obiecte ale aplicatiei). Dintre ei, setul de metode public disponibile constituie, intr-o exprimare intuitiva, „ceea ce stie sa faca obiectul” (echivalentul butoanelor de pe carcasa unui aparat)
- separarea interfeței de implementare: un obiect va putea fi manipulat in același fel, chiar daca modul in care își gestionează datele intern s-a schimbat. Si in viata reala, un aparat de radio poate avea un alt tip de potențiometru sau i se poate schimba complet placa de circuite, insa vom putea utiliza aparatul in continuare la fel ca pana acum cat timp butoanele de pe carcasa raman aceleasi si isi pastreaza functiile. Programatorul poate modifica oricat de mult functionarea interna a obiectului atata timp cat interfata publica a obiectului cu utilizatorii sai nu se schimba

Exemplu:

<pre> class Catzel { private int nrPurici, nrFirePar; public void scarpinare() { nrPurici--; nrFirePar-=10; } public int catiPurici() { return nrPurici; } } </pre>	<pre> class Catzel { // un catel neingrijit private long nrPurici; // ...si cam golas private short nrFirePar; // cati purici cad la o scarpinare byte puriciPerScratch=1; public void scarpinare() { nrPurici-=puriciPerScratch; nrFirePar-=10; } public int catiPurici() { return (int)nrPurici; } } </pre>
--	---

Avem aici doua implementari diferite ale clasei *Catzel*; daca un programator ce folosea prima varianta a clasei *Catzel* in aplicatia sa o inlocuieste cu cea de-a doua, codul deja scris nu are de suferit, deoarece obiectele *Catzel* au in ambele cazuri aceeasi interfata cu exteriorul (metodele *scarpinare()* si *catiPurici()*), care vor avea același efect, întorcând același rezultat scontat. Altfel spus, interfata obiectului cu exteriorul nu s-a schimbat, ci doar implementarea interna, care este ascunsa celui ce utilizează obiectul. In acest fel, utilizatorul este decuplat de aspectele interne ale functionarii obiectului, iar autorul clasei are un mare grad de libertate in a modifica implementarea ei interna.

3.4.2. Modificatori si niveluri de acces

Sa ne amintim ca definitia unui membru al clasei includea, printre altele, un asa-numit modificator de acces. Iata modificariorii si nivelurile de acces existente in Java, impreuna cu efectul acestora:

Modificator de acces	Nivel de acces rezultat	Efect asupra vizibilitatii membrului pentru utilizatorii externi ai obiectului
public	public	DA
private	private	NU
protected	protected	DA, insa doar daca accesarea se face din cadrul aceluiasi pachet ¹
(lipseste)	default (package-private)	DA, insa doar daca accesarea se face din cadrul aceluiasi pachet

Nota: nivelurile de acces protected si default nu sunt identice; efectul lor este acelasi doar in privinta vizibilitatii unui membru din afara obiectului, insa difera cand vine vorba de relatii de mostenire inter-clase (prezentate intr-un capitol ulterior al acestui curs)

Iata un exemplu care acopera toate cazurile posibile; pentru simplitate el foloseste doar campuri, insa concluziile sunt valabile in mod egal si pentru metode:

Clasa A	Clasa B aflata in acelasi pachet cu A	Clasa C aflata in alt pachet
<pre>class A{ public int x; private int y; protected int z; int t; void f(){ x++; // ok y++; // ok z++; // ok t++; // ok } }</pre>	<pre>class B{ void f(){ A a1 = new A(); a1.x++; // merge a1.y++; // NU merge a1.z++; // merge a1.t++; // merge } }</pre>	<pre>class C{ void g(){ A a2 = new A(); a2.x++; // merge a2.y++; // NU merge a2.z++; // NU merge a2.t++; // NU merge } }</pre>

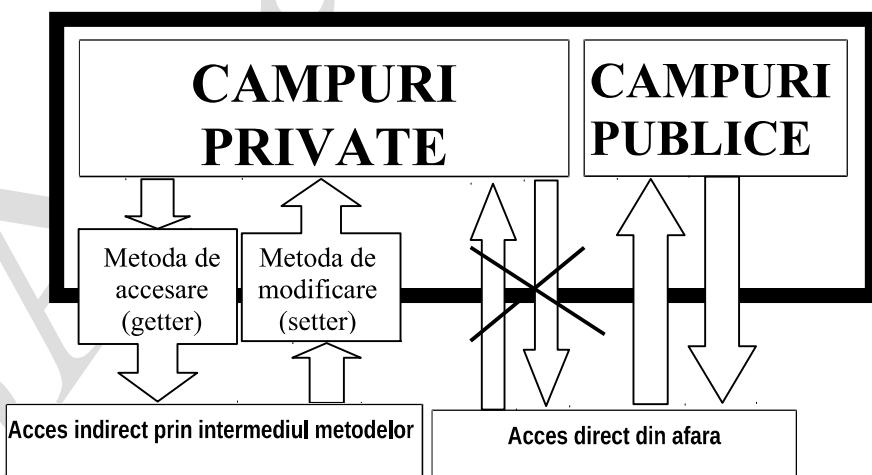
Nota: un membru al clasei este intotdeauna vizibil in metodele clasei, indiferent de nivelul sau de acces.

A se remarca faptul ca totul se reduce in final la *public* si *private*; in mod firesc, un membru poate fi vizibil sau nu - nu exista alta posibilitate. Putem privi celelalte doua niveluri de acces ca fiind un fel de *public/private* conditionat de context.

3.4.3. Metode de tip getter si setter

Odata ce campurile sensibile au fost ascunse in interiorul obiectului setandu-le un nivel de acces mai restrictiv decat *public*, apare problema accesarii si modificarii lor. Ele nemaifiind accesibile direct (ceea ce ne si doream), avem nevoie de niste metode care sa intermedieze accesul la ele. Metodele de citire/scriere se numesc astfel:

- **setter** sau **mutator** - metoda care modifica valoarea unui camp al obiectului
- **getter** sau **accessor** - metoda care citeste valoarea unui camp din obiect



Atat getterul, cat si setterul sunt metode per-camp: pentru fiecare atribut privat al unei clase se creeaza - daca este nevoie - propriul getter si/sau setter. Niciuna dintre aceste metode speciale nu este obligatorie - iata situatiile posibile:

- vom crea doar getter pentru un camp a carui valoare nu se va modifica de catre utilizatorul obiectului
- vom crea doar setter pentru un camp a carui valoare nu trebuie sa fie disponibila utilizatorului obiectului, dar pe care acesta o poate modifica (situatie mai rara)
- vom crea atat getter cat si setter pentru un camp care va fi atat accesat, cat si modificat de catre utilizatorul obiectului

1 Un pachet Java reprezinta un grup de clase care, gracie nivelului de acces default, au relatii mai stranse intre ele decat cu clase din afara pachetului

- nu este nevoie sa cream nici getter, nici setter pentru campuri care tin de “bucataria internă” a obiectului si deci nu sunt accesibile utilizatorului acestuia

Convenția de formare a definitiei acestor metode speciale este urmatoarea:

- setterul:
 - are numele format din *set* urmat de numele campului pentru care a fost creat (ex: pentru un camp *varsta*, setterul se va numi *setVarsta()*)
 - nu are date de intoarcere (tip de date produs *void*)
 - primeste ca argument noua valoare ce se doreste introdusa in obiect
- getterul
 - numele sau difera in functie de tipul de date al campului pentru care este scris:
 - pentru campurile de tip boolean, numele getterului este format din *is* si numele campului (ex: pentru un camp *married*, getterul se va numi *isMarried()*)
 - pentru campurile de orice alta natura, numele este format din particula *get* si numele campului (ex: pentru campul *varsta* avem getterul *getVarsta()*)
 - tipul de date de intoarcere coincide cu cel al campului pentru care a fost scris
 - nu are argumente

Avantajul este ca putem astfel filtra accesul la campurile private, iar modificarea lor se poate face intr-un cadru bine controlat (metoda de modificare efectueaza validarea datelor înainte de a le atribui unui câmp privat). Spre exemplu, clasa urmatoare incapsuleaza numere naturale, iar la incercarea de modificare a valorii verifică dacă numărul este pozitiv (tipul de date folosit pentru stocare este **int**):

```
class NumarNatural {
    // x stochează un număr natural; îl memorăm într-un întreg, și avem grija la atribuire
    private int nrNatural;

    public int getNrNatural() { return nrNatural; }
    public int setNrNatural(int n) {
        if (n>=0)
            nrNatural=n;
        else
            System.out.println("Eroare! "+n+" este negativ!");
    }
}
```

3.5. Constructorul

Cu elementele prezentate pana in acest moment putem crea clase care obiecte incapsuleaza intotdeauna date valide conform cerintelor aplicatiei, gratie procedeului de incapsulare. In practica constataam insa urmatoarele neajunsuri:

- pentru a crea doua obiecte ale aceleiasi clase care au stare (continut) diferita, este necesar ca mai intai sa le instantiem - ele vor primi in prima faza valori ale atributelor identice - si abia apoi sa stabilim valorile dorite ale atributelor (fie prin acces direct, daca acesta este permis, fie prin intermediul metodelor de tip setter). Ar fi mai eficient sa putem crea obiectul de la bun inceput cu continutul dorit, nu in doua etape
- ori de cate ori este posibil, in Java este practic sa se lucreze cu obiecte *immutable*: odata obiectul creat, continutul sau nu mai poate fi modificat. Este simplu de realizat un astfel de obiect: in definitia clasei nu prevedem metode de tip setter, iar atributele sale vor avea nivel de acces *private*. Apare insa problema: cum initializam atunci continutul obiectului cu valorile dorite???

Raspunsul la ambele probleme este **constructorul** - o pseudo-metoda cu regim special, executata automat la instantierea obiectului. Mai precis, constructorul se executa dupa ce operatorul *new* aloca memorie pentru noul obiect si campurile sunt initializate cu valorile lor default sau specificate explicit, insa inainte ca *new* sa returneze referinta catre noul obiect creat. Constructorul reprezinta modalitatea de a executa cod cu ocazia instantierii; codul poate folosit pentru a initializa campurile obiectului (in cele mai dese cazuri) dar si in alte scopuri.

Caracteristicile constructorului sunt:

- se aseama cu o metoda din urmatoarele puncte de vedere:

- are nume si posibila lista de argumente
- are un corp delimitat de { si } in care pot fi utilizate instructiuni Java si in care sunt vizibile campurile clasei (de aceea poate fi folosit cu succes pentru initializarea campurilor)
- poate fi supraincarcat
- difera de o metoda obisnuita prin urmatoarele proprietati:
 - nu are tip de date de intoarcere (semnatura sa nu contine DELOC acel camp, nici macar void!)
 - are nume impus, identic cu al clasei
 - nu poate fi apelat dintr-o alta metoda a clasei sau din afara acesteia ca o metoda obisnuita (folosind numele sau si lista de argumente), ci doar in urmatoarele moduri:
 - indirect, instantiind un obiect al clasei (fie din cadrul clasei, fie din afara ei)
 - direct, din cadrul altui constructor al aceleiasi clase, insa cu o sintaxa speciala (vezi mai jos)
 - nu se mosteneste in subclase (vezi capitolul dedicat mostenirii)

Atunci cand constructorul are argumente, sintaxa de instantiere contine valori pasate constructorului, asemanator cu apelarea unei metode:

```
class Catzel {
    String nume;
    Catzel(String n) {
        nume=n;
        System.out.println("S-a creat un obiect Catzel cu numele "+n);
    }
}
class Prog1 {
    public static void main(String[] args) {
        Catzel c=new Catzel("Azor"); // S-a creat un obiect Catzel cu numele Azor
    }
}
```

Orice clasa are un constructor. Acest lucru se poate realiza pe doua cai:

- daca programatorul nu definește nici un constructor, compilatorul va insera automat in bytecode așa-numitul **constructor din oficiu (default)**, care are corp vid si nu are argumente.
- daca programatorul defineste cel puțin un constructor, **cel default nu va mai fi adaugat!** Consecinta este ca, daca programatorul a definit un constructor cu argumente, instantierea nu se va mai putea face fara ele! Spre exemplu, daca pentru clasa *Catzel* de adineauri am incerca:

```
Catzel c = new Catzel();
```

am primi o eroare deoarece nu exista un constructor cu acea semnatura (singurul disponibil are argument).

Constructorul clasei poate fi suprainscris (pot exista mai mulți constructori, care primesc seturi de argumente diferite); in aceste conditii, **new** poate fi apelat folosind oricare din variantele de constructor. Atunci cand este nevoie de refolosire de cod prin apelarea unui constructor din cadrul altuia, se va folosi in acest scop cuvantul cheie *this* ca in exemplul urmator:

```
class Catzel {
    private String nume;
    private int nrPurici;

    Catzel(String n,int nr) { nume=n; nrPurici=nr; }
    Catzel(String n) { this(n,0); } // pentru catei "curati"
    Catzel() { this("Canionimul",0); } // instantiere fara argumente; toate
} // campurile capata valori default

public class Haita {
    public static void main(String[] args) {
        Catzel c1=new Catzel(),
               c2=new Catzel("Azor"),
               c3=new Catzel("Labush",14);
    }
}
```

Atentie! Atunci cand apelam un constructor al clasei din cadrul altuia, apelul *this* trebuie sa fie prima instructiune din constructorul apelant!

Se naste intrebarea: cand am vrea sa folosim initializarea unui camp la declarare si cand pe cea din constructor? Raspunsul este simplu: prima dintre ele este „hard-coded” (la instantierea fiecărui obiect va fi folosita aceeași expresie/valoare **indiferent de instanță**), pe cand in cadrul unui constructor care primește parametri se poate face o initializare **specifică unei anumite instante**.

3.6. Distrugerea obiectelor

Spre deosebire de alte limbaje, unde programatorul era obligat sa elibereze explicit, prin intermediul destructorului, orice memorie alocata la crearea unui obiect, in Java responsabilitatea eliberarii memoriei este luata de pe umerii programatorului si data lui **garbage collector (gc)**, un fir de executie de prioritate scazuta al masinii virtuale care elibereaza memoria heap alocata cu **new**. Garbage collector analizeaza fiecare obiect din heap si elibereaza memoria ocupata de catre acesta numai in cazul in care nu mai exista referinte catre obiect din nici unul dintre firele de executie active ale programului.

Programatorul nu are control asupra momentului in care ruleaza garbage collector; *gc* intervine atunci cand masina virtuala se apropie de limita superioara a memoriei disponibile – ceea ce este posibil sa nu se intampla niciodata in cursul executiei daca aplicatia nu este mare consumatoare de memorie – sau in urma invocarii explicite prin intermediul apelului **System.gc()**. Atentie insa! Simpla invocare nu garanteaza faptul ca garbage collector va rula sau ca va dezaloca toata memoria ocupata de obiecte „moarte”! Sa nu uitam ca el este un fir de executie de prioritate mica, si in aceste conditii este posibil ca el sa nu ruleze deloc sau sa elibereze memoria intr-un ritm mai lent decat ne-am dorit.

Nota: ca o consecinta a acestui principiu de functionare, un obiect poate ramane in memorie chiar si dupa ce ultima referinta catre el inceteaza sa existe (de exemplu, in cazul unei variabile locale, aceasta dispare de pe stiva la iesirea din blocul/metoda in care a fost definita)

Daca dorim ca un obiect sa devina eligibil pentru garbage collecting inainte ca variabilele care se refera la el sa dispara „natural”, putem atribui acestora in mod explicit valoarea **null** odata ce nu mai folosim obiectul in cauza.

Fiecare obiect posedă o metoda *finalize()*, care este apelata automat imediat inaintea distrugerii acestuia. Ea nu este echivalenta cu destructorul din C si nu este folosita pentru dezalocarea memoriei, ci ii ofera programatorului ocazia de a elibera resursele de alta natura folosite de catre obiect (ex: inchidere de fisiere sau de conexiuni).

Atentie!

1. Nici garbage collector, nici metoda *finalize()* nu reprezinta un echivalent al destructorului din C
2. Implementarea metodei *finalize()* poate fi de asa natura incat sa impiedice stergerea obiectului (de exemplu, prin memorarea intr-o variabila din program a unei noi referinte catre obiectul curent). Oricum ar fi, **metoda finalize() se executa insa o singura data!**
3. Garbage collector stie sa elimine si „insulele” de obiecte (grup de obiecte care au referinte unul catre altul insa *nu mai sunt referite din nici un fir de executie activ*)

3.7. Relatia obiecte virtuale - obiecte reale

Realitatea este si ea un ansamblu de obiecte care interactionează, ceea ce face mai usoara trecerea din limbaj uman intr-un limbaj de programare orientat pe obiecte. Care este insa regula după care se definesc obiectele in Java? Nu exista una universală. Cand scriem o clasa Java definim un tip de date – un set de caracteristici si manifestari comune tuturor obiectelor ce se vor instanta.

Un prim aspect este identificarea celor doua categorii de membri ai clasei: pentru a modela un obiect real, trebuie sa ne clarificam:

- ce “știe sa facă” obiectul, cum interacționează cu alte obiecte – ce informație schimba, cum reacționează la mesajele primite, care sunt metodele care trebuie să fie disponibile public (interfață)
- ce proprietăți (attribute) reale are acel obiect, care dintre ele sunt relevante în aplicația noastră și cum se pot ele exprima în termeni de tipuri de date. Acestea rezultă în bună parte din ce stie să facă obiectul

Un obiect virtual se definește în funcție de necesitățile aplicației – el conține doar acele caracteristici ale obiectului real care ne sunt utile (doar datele ce vor interveni în prelucrare). De exemplu, o biblioteca virtuală va conține obiecte de tip carte, care pot fi definite astfel:

```
public class Carte {  
    String gen, titlu, autor;  
    int numarInventar;  
}
```

Un obiect virtual Carte va avea numai gen, titlu, autor și număr de inventar; deși o carte reală are și format, număr de pagini, pret etc., este posibil ca ele să nu fie necesare aplicației și deci nu vor face parte din obiect.

3.8. BIBLIOGRAFIE

- Java tutorial
 - clase și obiecte
 - <http://docs.oracle.com/javase/tutorial/java/javaOO/classes.html>
 - <http://docs.oracle.com/javase/tutorial/java/javaOO/objects.html>
 - <http://docs.oracle.com/javase/tutorial/java/javaOO/more.html>