Note de Curs V.1.0

PROGRAMAREA ORIENTATĂ OBIECT

Sintaxa JAVA

By Melnic Nicolae

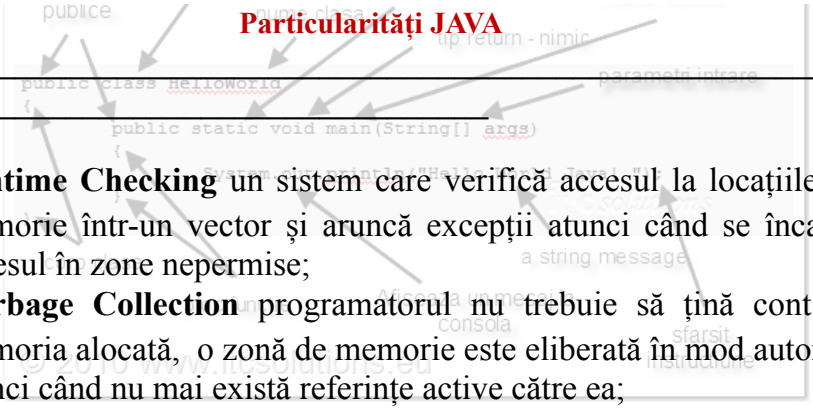
2014

Pe Scurt Despre JAVA

- Java este un limbaj de programare **orientat obiect(nu este pur OO)**;
- Dezvoltat la **Sun Microsystems, Inc**;
- Conceput de către **James Gosling**;
- În anul **1995**;
- Este un limbaj de programare **key sensitive**;
- Este un limbaj de programare foarte **sigur**, având mecanizme de securitate stricte;
- Java este rulata pe **platforma independenta**;
- Programele Java sunt **compilate și apoi interpretate**;
- Ruleaza de pe platforma **JVM(Java Virtual Machine)**;
- Se folosește la programarea aplicațiilor pentru **dispozitivile mobile gen telefon, etc**;
- **Cea mai mare parte a aplicațiilor** sunt scrise în java;
- **Înprumută o mare parte din sintaxa de la C și C++** (doar că are un model al obiectelor mai simplu);
- Medii de dezvoltare a aplicațiilor java intergrate(**Jcreator, Eclipse, NetBeans, Jbuilder, Jdeveloper, Kdeveloper, CodeGuide, etc.**);
- Toate distribuțiile Java sunt oferite **gratuit;**

Avantajele programării în Java

- ✓ **Java este gratuit.** Nu numai mașina virtuală Java (*JVM*) și pachetul pentru dezvoltatorii de programe JDK(*Java Development Kit*) sunt gratuite, dar și există și o sumedenie de medii de dezvoltare intergrate(*IDE*), pentru Java care sunt free sau open source.
- ✓ **Java este portabil.** Datorită rularii programelor prin intermediul unei ”Mașini Virtuale”(Java Runtime Environment), aceasta sunt independente de arhitectura hardware a calculatorului și de sistemul de opere folosit.
- ✓ **Java este ușor de învățat.** Limbajul Java are la bază mai vechiul C/C++, de la care moștenește elementele esențiale, dar din care au fost scoase acele caracteristici care îngreunau programarea sau faceau programele confuze și instabile. Astfel Java posedă aceleași facilități puternice de programare orientată obiect (POO) și cam aceeasi sintaxă ca si C++, dar este mult mai ușor de utilizat;
- ✓ **Java este ideal pentru Internet.** Acest lucru se datorește faptului că a fost creat special pentru a lucra în rețea și mediul online fiind foarte potrivit pentru realizarea aplicațiilor distribuite pe internet.
- ✓ **Aplicațiile Java sunt ușor de utilizat.** Datorită faptului că programele Java sunt de fapt niște colecții de clasa compilate sub forma ”Codului de Octeți”(ByteCode), care va fi interpretat de mașina virtuală lacală, distribuția lor nu necesită instalare și programe complexe de setup, care ar necesita un efort din partea utilizatorului pentru a le folosi și a le întelege.
- ✓ **Programele Java au dimensiuni mici.** Deși codul este compilat, totuși instrucțiunile sale sunt adresate interpretorului Java, care se ocupă de funcțiile sistem și interacțiunea cu componentelor hardware. În consecința programelor Java ocupă mult mai puțin spațiu pe disc în comparație cu cele ”clasice”.



- **Runtime Checking** un sistem care verifică accesul la locațiile de memorie într-un vector și aruncă excepții atunci când se încarcă accesul în zone nepermise;
- **Garbage Collection** programatorul nu trebuie să țină cont de memoria alocată, o zonă de memorie este eliberată în mod automat atunci când nu mai există referințe active către ea;
- **Distributed Coputing –** java pune la dispoziție în bibliiotecile implicite clare care faciliteaza conectivitatea (java.net) și execuția distribuită java.rmi);
- **Multihreading** – java pune la dispoziție în bibliotecile implicite clare care facilitează execuția concurentă a mai multor metode și sincronizarea acestora;

În plus față de **aplicațiile de sine stătătoare (standalone)**, în Java există **suport** pentru un tip de aplicații numite applet-uri, care se execută în interiorul unui browser de web. Acest lucru facilitează dezvoltarea aplicațiilor interactive pe web.

Totuși, poate cel mai important avantaj al limbajului Java, este **setul de clase puse la dispoziție de Oracle**, numit generic **Application Programming Interface**, care conține o vastă **colecție de funcții** deja implementate, gata de a fi folosite pentru o gamă largă de aplicații.

Programarea Orientată Obiect

POO – este un paradigma(model) de programare utilizind obiecte.

Sau:

Este o metoda de programare bazata pe clase si erarhii, si coopereaza intre obiecte.

Sau:

Este o metodologie de programare pentru a proiecta soft-uri utilizind clase si obiecte.

<u>Notiuni:</u>

- Este unul **din cei mai importanți pași facuți în evoluția limbajelor de programare**;
- A apărut din necesitatea **exprimării problemei într-un mod mai natural ființei umane**;
- **Ideia POO este de a crea programe ca o colecție de obiecte**, unități individuale de cod care interacționează unele cu altele, în loc de simple liste de instrucțiuni sau de apeluri de proceduri;
- **Obiectele POO sunt de obicei reprezentări ale obiectelor din viața reală**, astfel programele realizate prin tehnica POO sunt mai ușor de extins și de depanat decât programele procedurale;
- **POO** este o metoda de implementare in care programele sunt organizate ca ansamble de obiecte ce coopereaza intre ele, fiecare obiect reprezentand instanta unei clase; fiecare clasa apartine unei ierarhii de clase in cadrul careia clasele sunt legate prin relatii de mostenire.

<u>Conceptele programarii orientate obiecte:</u>

OBIECTE | CLASE| ABSTRACTIZARE | INCAPSULARE | MOSTENIRE | POLIMORFISM.

Avantajele POO

- ✓ Codul devine mai lizibil, mai ușor de depanat;
- ✓ Putem crea programe mari foarte ușor, reușind să gândim codul corect încă de la început;
- ✓ Prin abstractizare putem grupa codul corespunzător unui concept și totodată îl putem separa de restul codului;
- ✓ Prin Incapsulare(ascunderea unor date pentru anumite entități) putem evita accesul necontrolat la date;
- ✓ Unul dintre avantajele oferite de tehnologia POO este posibilitatea de a extinde comportamentul unei clase existente prin definirea unei clase noi care mosteneste continutul primei clase, adaugand la acesta elemente specifice.
- ✓ Un avantaj al POO este conceptul de mostenire/derivare pentru ca iti permite sa definesti clase noi pe baza unor existente, prin extinderea acestora si nu prin modificarea lor.
- ✓ Un principiul al programarii structurale este reutilizarea codului. Un program orientat pe obiect nu este niciodata scris de la zero, ci foloseste clase scrise de alti programatori.

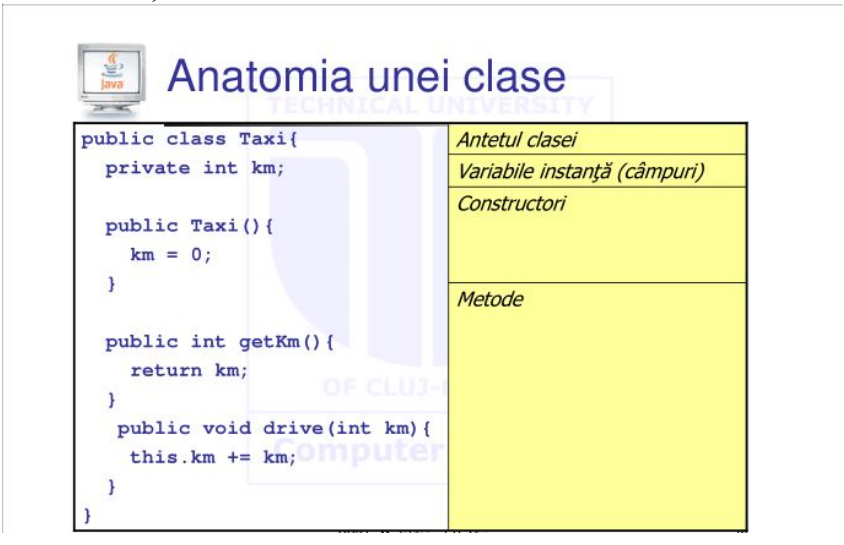
Principiile Programării Orientate Obiect

Clase

O clasa este o structura de date compacta, care contine variabile si functii(numite metode) prin care se pot crea diferite instructiuni legate intre ele in acea structura.

Clasele reprezinta principiile de baza ale programarii orientate obiect. Clasele reprezinta notiuni abstracte, povesti, modele (blueprints), ce descriu:

- caracteristicile, atributele unui obiect; reprezinta ceea ce stie sau ce este obiectul;
- ce stie sa faca obiectul, comportamentul acestuia descris de metodele sale;



Notiuni:

- ✓ O clasă este o descriere a obiectelor (instanțelor), clasei respective;
- ✓ O clasă este o structură care poate contine atât date cât si functii ca membri ai structurii;
- ✓ Grupează datele şi unităţile de prelucrare a acestora într-un modul, unindule astfel întro entitate mult mai naturală
- ✓ O clasă conține unul sau mai mulți constructori pentru a crea obiecte.
- ✓ o clasă are un corp (care conţine atribute şi/sau metode) definite între { şi };

- ✓ clasa care conţine funcţia **main** are acelaşi nume (la nivel de litera – case sensitiv) cu fişierul sursa; clasa *HelloWorld* este definita în fisierul *HelloWorld.java;* dacă modificaţi numele clasei obtineti la compilare eroarea:
- ✓ o clasă reprezintă un tip de date definit de utilizator, care se comportă întocmai ca un tip predefinit de date. Pe lângă variabilele folosite pentru descrierea datelor, se descriu şi metodele (funcţiile) folosite pentru manipularea lor.
- ✓ Clasa este un sablon folosit la crearea(instantierea) tuturor obiectelor de acel tip(din acea clasa).
- ✓ Clasa defineste structura obiectului de acel tip.
- ✓ Clasa defineste comportamentul tuturor obiectelor de acel tip. Clasa = structura + component.

Instanţa unei clase reprezintă un obiect - este o variabilă declarată ca fiind de tipul clasei definite. Variabilele declarate în cadrul unei clase se numesc variabile membru, iar funcţiile declarate în cadrul unei clase se numesc metode sau functii membru. Metodele pot accesa toate variabilele declarate în cadrul clasei, private sau publice.

Instantele unei clasa sunt obiecte:

Se creaza un obiect de tipul Student, adica se instantiaza un obiect din clasa Student:
Class Student {
Instanta student;
O alta instanta Student;
}

Definitia unei clase este compusa din:

- ✓ Declaratia clasei - (este obligatorie; declara existenta unui nou tip de date).

✓ Implementarea clasei – (poate fi partiala; specifica detaliile clasei: structura si comportamentul).

Exemplu:

Definitie = declaratie + implementare;

Orice declaratie de clasa contine cel putin:

✓ Cuvintul cheie ***class***
✓ *Numele clasei*

Exemplu:

```
Class Student {
//*Corpul clasei
}
```

✓ Dupa declaratia clasei urmeaza obligatoriu parantezele acolada.
✓ Nu se pune punct si vigula dupa definitia clasei.
✓ Intre acolade este corpul clasei(implimentarea).
✓ Cuvintul cheie class poate fi precedat de modificatorul de acces public.
✓ Modificatorul de acces specifica cine paote folosi clasa resprectiva.
✓ O clasa declarata publica trebuie sa fie scrisa intr-un fisier cu numele clasei.
✓ Clasa Student trebuie salvata in fisierul Student.java.
✓ Drept consecinta, un fisier sursa Java contine cel mult o clasa publica, iar aceasta are numele fisierului.
✓ Un fisier sursa Java poate contine una sau mai multe clase cu modificatorul de acces implicit. Clasele respective pot fi utilizate doar in clasa publica a fisierului sau in orice alta clasa din acelasi pachet de clase, dar nu in afara acelui pachet de clase. Ele sunt

clase ajutatoare, folosite la implementarea claselor publice ale acelui pachet.

Implementarea clasei:

- ✓ Implementarea clasei este descriere structurrii si comportamentului acesteia.
- ✓ Structura este data de variabilele clasei.
- ✓ Comportamentul este defini prin functii.

Exemplu:

implementare = variabile + functii;

- ✓ Elementele descrierii clasei, adica variabilel si functiile, se numesc membri ai clasei.

Membrii unei clase:

Membrii unei clase reprezintă totalitatea metodelor şi a variabilelor membre ale clasei.

O clasa in Java este definita de regula prin:

- ✓ Campuri - (numite si atribute sunt variabile de stare. Un camp poate fi: o variabila de tip primitiv(int, float, char), o referinta la obiect(String), un tablou de variabile sau de referinta(int[], String[]).
  Numele unui cimp incepe cu o litera mica, daca numele este compus, cuvintele care urmeaza incept cu litera mare sau sunt despartite de caracterul"_" si nu se admit spatii in numele cimpului(exemplu de mai jos).
  Declaratia unui cimp este formata din tipul variabilei sau tipul

obiectului de referinta urmat de numele cimpului).

Exemple Campuri:

Int nr_credite;
String nume;
Int notaExamen[]; sau int [] notaExamen;
String numeExamen[]; String[] numeExamen;

- ✓ Constructori;
- ✓ Metode;

Animal
nume : *String;*
tip : *String;*
varsta *: int;*
getNume() : String;
setNume(nume : String) : void;
getTip() : String;

O clasa este reprezentata ca un dreptunghi cu trei zone care contin:

- ✓ Numele clasei
- ✓ Atributele clasei
- ✓ Metodele clasei.

Sintaxa generală declarării unei clase este următoarea:

```
[modificator_acces] class nume_clasa [extends base_class]
[implements interface1, interface2, …]

{//inceput bloc clasa

//atribute

//metode

//blocuri de cod
```

```
//alte clase


}//sfarsit bloc clasa
```

Pe baza sintaxei simple, se defineste o clasa simpla Java ce descrie o carte. Cartea este descrisa de atributele *pret, titlu* si *autor*, iar comportamentul este definit de metodele *getPret()* si *afiseaza()*.

```
class Carte{
    //definire atribute - variabile pentru instante
    float pret;
    String titlu;
    String autor;

    //definire metode
    public float getPret(){
        return pret;
    }
    public String afiseaza(){
        return "Cartea "+titlu+" are ca autor "+autor;
    }
}
```

Avantajele Claselor:

- ✓ Codul devine mai lizibil mai usor de înțeles și depanat;
- ✓ Putem crea programe mai foarte ușor, reușind să gindim codul corect încă de la bun început;
- ✓ Prin abstractizare putem grupa codul corespunzător unei concept și totodată îl putem separa de restul codului;
- ✓ Prin incapsulare(ascunderea unor date pentru anumite entități) putem evita accesul necontrolat la date;
- ✓ Putem modela relațiile dintre entități;
- ✓ Putem implementa un mod de comunicare între entități.

Clase Fundamentale:

- ✓ **Clasa System** - face parte din pachetul java.lang, nu are constructor public.

- ✓ **Clasa String** – este declarata final, si toate atributele ei sunt final. Un obiect String odata ce a fost creat nu mai poate fi modificat. Nu

compara doua siruri ci doua referinte, petntru a compara doua siruri se foloseste metoda equals sau compareTo. Concateneaza doua siruri. Toate tipurile primitive pot fi convertite la tipul String. Conversia nu se face automat decit la concatenarea ca un sir. Orice obiect care are metoda toSting poate fi convertita la String.

- ✓ **Clasa Math** – face parte din java.lang, are atributele statice(E, PI), are metode statice(max, min, sqrt, random), functii trigonometrice, etc.

- ✓ **Clasa Random** - face parte din java.util, are constructor cu si fara argumente, are metode.

Clase abstracte:

- ✓ Cu cit sunt mai sus intr-o erarhie de clase, superclasele devit tot mai generale si abstracte.
- ✓ La un moment dat unele superclase sunt mai degraba un model pentru alte clase decat o definite a unor obiecte.
- ✓ In acest caz superclasele sunt declarate abstracte si nu se pot crea obiecte din ele.
- ✓ metoda abstracta este o metoda fara corp(continut).
- ✓ clasa care contine o metoda abstracta, trebuie declarara clasa abstracta.
- ✓ Nu se poate crea obiecte din clasa abstracta.
- ✓ clasa poate fi declarata abstracta si daca nu contine nici o metoda abstracta.

Clase finale:

- ✓ Final aplicat la o variabila sau atribut il face constant; dupa ce a primit o valoare nu mai poate fi modificat.

- ✓ Final aplicat la o metoda face ca metoda sa nu mai poata fi redefinita.
- ✓ Final aplicat la o clasa face ca sa nu se mai poate deriva alte clase din ea.

Obiecte

Obiect – componenta software care incorporeaza atit atributele cat si operatiile(metode, functii) care se pot efectua asupra atributelor si care suporta mostenirea.

Sau - Blocuri de constructie a sistemelor software.

Sau - este în deobicei o componentă dintr-un program sau o aplicaţie, alături de alte obiecte, existenţa singulară a unui obiect nefiind prea folositoare. Tot secretul metodei orientate pe obiecte constă în abilitatea de a face diferite obiecte, fiecare descriind o anumită entitate, să interacţioneze între ele şi în urma producerii acestora să se obţină rezultatele scontate. (exemplu: existenţa unui obiect “carte” nu are sens decât dacă există un alt obiect care să extragă informaţii din “carte”, apelând o metodă de genul “citeşte()").

Notiuni:

- ✓ Lumea reala este un ansamblu de obiecte care interactioneaza
- ✓ Orice sistem poate fi privit ca un ansamblu de obiecte care interactioneaza.
- ✓ Un obiecte este o entitate ce poate fi descrisa independent de ansamblu din care face parte.

- ✓ Obiectele poat fi concrete, fizice, cu existenta reala independenta de mintea nostra, cele pe care le putem atinge sau percepe direct print simturile noastre, sau abstracte, virtuale create doar in imaginatia noastra.
- ✓ Gandirea noastra este deseori (se poate spune chiar intotdeauna), si fara sa ne dam seama, orientata pe obiecte.
- ✓ Naturaletea si realismul modelelor bazate pe obiecte au facilitat adoptarea abordariii obiect-orientate a celor mai diverse aplicatii.
- ✓ Orice program orientat pe obiecte este conceput ca un sistem de obiecte care interactioneazza prin intermediul unor proceduri adecvate tipurilor de obiecte, stabilite de insasi natura obiectelor.
- ✓ Obiectele soft, ca şi obiectele din lumea reală, au o viaţă limitată. Ele sunt create (îi sunt alocate resurse), folosite şi distruse (resursele alocate sunt eliberate).

În jurul nostru/ in lumea reală putem găsi o mulțime de **obiecte:** masini, case, oameni, animale, etc. Toate aceste pot avea forma și comportament.

Exemplu:

Cîinele are cap, picioare, coadă, blană şi poate alerga, dormi, lătra, etc. Aceste la rindul lui este un obiect, care are caracteristici si propreitati dupa cum am vazut anterior. Toate aceste obiecte si caracteristici sunt componentele de bază a unei clase.

Descrierea obiectelor:

Orice obiect poate fi caracterizat sau descris prin:

1. ***Pozitie –*** identifica obiectul in cadrul sistemului din care face parte.
2. ***Stare –*** caracterizeaza structura obiectului la un moment dat.
3. ***Comportament -*** obiectul descrie raspunsul acestuia la stimuli

externi si la modificarea starii. Comportamentul obiectului depinde de starea in care se afla.

Orice obiect poate fi descris prin:

- ***Structura***
- ***Comportament.***

Clasa defineste structura si comportamentul obiectelor din clasa:

Clasa = structura + comportament

Pentru a descoperi continutul unui obiect este util sa punem doua intrebari:

- Ce stie obiectul (ce cunoaste) – descoperim datele, atributele obiectului.
- Ce stie sa faca obiectul – descoperim comportamentul, metodele obicectului.

Accesul la membrii unui obiect:

- ✓ Accesul la un obiect se face prin intermediul unei referinte la acel obiect.
- ✓ Accesul la un camp(vizibil) al obiectului: *numeReferinta.numeCamp*
- ✓ *Apelul unei metode(vizibile) a obiectului: numeReferinta.numeMetoda();*
- ✓ *Membrii care nu sunt vizibili nu pot fi accesati;*
- ✓ *Valoare unui camp poate fi mofificate.*

Exemple de acces la membrii unui obiect:

- ✓ Accesul la un camp al unui obiect *Student*: ***student_1.nr_credite;***
- ✓ Accesul unei metode specifice unui obiect *Student:* ***student_2.setCredite(60).***
- ✓ Apelul unei metode specifice unui obiect *String*, obiect membru a unui obiect *Student:* ***student_1.nume.length().***
- ✓ Includere unor obiecte in alte obiecte se face prin intermediul referintelor *.nume* este o referinta la obiectul de clasa *String* din interiorul obiectului *student_1*.

Notiuni:

- ✓ Obiectele pot fi grupate, clasificate, ordonate in clase.
- ✓ Clasa de obiecte defineste structura (arhitectura, planul) obiectelor ce pot fi create din acea clasa.
- ✓ Obiectele sunt instante ale unor clase.
- ✓ Nu exista obiecte fara clase.
- ✓ Obiectele sunt structuri de date.
- ✓ Orice program scris in Java este un sistem de obiecte.
- ✓ Un obiect din program exista din momentul in care este creat(instantiat) pana cind este sters din memorie(Garbage colector)
- ✓ Obiectul exista din momentul instantierii pana cind se pierde orice referinta la acest obiect.
- ✓ In lipa unei referinte obiectul devine inutilizabil si paote fi sters din memorie

Diversitatea obiectelor:

- ✓ Obiectele pot fi la fel sau diferite,mai mult sau mai puti asemanatoare.
- ✓ Obiectele pot fi simple sau complexe.
- ✓ Obiectele pot fi la rindul lor compuse din obiecte.
- ✓ Obiectele pot ava un comportament simplu sau foare complicat.

✓ Obiectele poate avea o existenta efemera sau de durata.

Obiectele sunt construite prin operatorul *new* care va apela functia constructor din clasa (cu sau fara parametrii):

Sintaxa de declararea unui obiect in Java:

Pentru a testa clasa *Carte* definita mai devreme vom construi o noua clasa publica, *TestCarte* (fisierul Java se numeste tot TestCarte.java):

```
public class TestCarte {
   public static void main(String[] args) {

      //definire referinta

      Carte carte1;      //are valoarea null

      Carte carte2 = null;

      //creare obiect
       carte1 = new Carte();

      carte2 = new Carte();
   }}
```

Obiectul ca si variabila reprezinta o referinta (pointer) ce gestioneaza o adresa din Heap. Prin intermediul acestei adrese avem acces la zona de memorie rezervata pentru obiect in care se gasesc valorile atributelor sale.

Prin definirea unui obiect se obtine o simpla referinta care are valoarea implicita *null*. Pentru a da valoare acestei referinte se construieste (instantiaza) obiectul prin *new*.

Relatii intre obiecte:

R. de dependenta:

- ✓ Un obiect foloseste alt obiect(depinde de alt obiect)
- ✓ Este cea mai generala relatie dintre obiecte.
- ✓ Celalalt obiect nu apare ca atribut al primului obiect ci e folosit doar in metode.

R. de asociere:

- ✓ Un obiect stie despre alt obiect.
- ✓ Nu exista relatie de icluziune intre obiecte.
- ✓ Celalalt obiect apare ca atribut al primului obiect.
- ✓ Roluri si multiplicitate.

R. de agregare:

- ✓ Un obiect contine alt obiect.
- ✓ Al doilea obiect este un atribut al primului obiect.

R. de mostenire:

- ✓ Un alt obiect este alt obiect.
- ✓ Clasa de baza de mai numeste clasa parinte.
- ✓ Clasa fiu mosteneste clasa parinte.
- ✓ Clasa fiu mosteneste atributele si metodele clasei parinte.
- ✓ Clasa fiu poate adauga atribute si metode noi (Extinde clasa de baza).

Metode

Definesc comportamentul obiectului. Metoda= functie=procedur

Sunt cunosctute ca si proceduri sau functii:

- ✓ Procedurile nu returneaza nici o valoare
- ✓ Functiile retureneaza valoarea.

Ce poate face o metoda:

- ✓ Schimba starea unui obiect.
- ✓ Raporta starea obiectului sau.
- ✓ Opera asupra numerelor, a textului, a fisierelor, graficii, paginilor web, hardware.
- ✓ Crea alte obiecte.
- ✓ Apela o metoda a unui alt obiect: *obiect.metoda(args)*.
- ✓ Apela o metoda aceiasi clasa: this.metoda(args).
- ✓ Se poate apela autorecursivitatea.

Declaratia unei metode – este formata din tipul rezultatului metodei, numele metodei si intre paranteze normale, declaratiile argumentelor metodei.

Sintaxa de declarare a metodei:

TipRezultat numeMetoda (tipArg1, numeArg1,...){};

Numele unei metode – contin numai litere si cifre, incep cu litera mica, daca cuvintul este compus, cuvintele care urmeaza incep cu litera mare sau sunt despartite de caracterul "_", nu se admit spatii in numele metodei(exact ca la declarare unei clase).Mai sus a fost declarat un exemplu de metoda.

Tipuri de metode:

Metode constructur

- ✓ Sunte metode speciale apelate atunci cind se creaza un obiect.
- ✓ Metodele de tip constructor au exact acelasi nume cu cel al clasei de obiecte si nu au tip de returnare(nici macar voir).
- ✓ Orice clasa are cel putin un costructor,
- ✓ Daca nu declaram nici un constructor, compilatorul va genera un constructor default.
- ✓ Constructorii sunt folositi pentru initializarea datelor de obiect.
- ✓ Petntru clasa ce nu are nici un constructor public, nu se poat crea obiecte din afara clasei.

Metode getter

- ✓ Sunt metode de tipul public int getNota().
- ✓ Unde nota este un atribut de tip intreg
- ✓ In mod normal atributele sunt private, iar petnru a avea acces in

mod controlat la atributele unui obiect se folosesc metode geter.

Metode setter

- ✓ Sunt metode de tip public void setNota(int nota).
- ✓ Unde nota este un atribut de tip intreg.
- ✓ In mod normal atributele sunt private, iar pentru a modifica in mod controlat atributele unui obiect se folosesc metode setter,
- ✓ In mod normal pentru toate atributele private care dorim sa poate fi scrise din afara obiectului, cream metoda setter.

Metode generale

- ✓ Sunt toate celelate tipuri de metode.
- ✓ Orice metoda primeste unele argumente si returneaza un rezultat.

Conceptele POO

Principalele concepte (caracteristici) ale POO sunt: ***Moștenirea, Polimorfismul, Încapsularea, Abstractizarea.***

✓ ***Moștenirea*** este procesul care permite unui obiect să preia proprietățile altui obiect.

Sau - posibilitatea de a extinde o clasa prin adaugarea de noi functionalitati.

Sau - a deriva noi definitii de clase din clase existente.

Sau – Organizează și facilitează polimorfismul și încapsularea,

permițând definirea și crearea unor clase specializate plecând de la clase (generale) deja definite - acestea pot împărtăși (și extinde) comportamentul lor, fără a fi nevoie de a-l redefini. Aceasta se face de obicei prin gruparea obiectelor în clase și prin definirea de clase ca extinderi ale unor clase existente. Conceptul de moștenire permite construirea unor clase noi, care păstrează caracteristicile și comportarea, deci datele și funcțiile membru, de la una sau mai multe clase definite anterior, numite clase de bază, fiind posibilă redefinirea sau adăugarea unor date și funcții noi. Se utilizează ideea: ”Anumite obiecte sunt similare, dar în același timp diferite”. O clasă moștenitoare a uneia sau mai multor clase de bază se numește clasă derivată. Esența moștenirii constă în posibilitatea refolosirii lucrurilor care funcționează.

Sau - Unul dintre marile avantaje ale programării orientate obiect este posibilitatea de a reutiliza codul existent. Posibilitatea proiectării de noi clase folosind clase deja construite se numeşte moştenire. Dacă o clasă A moşteneşte o clasă B, atunci variabilele şi metodele clasei B vor fi considerate ca aparţinând şi clasei A. Moştenirea permite crearea unor clase “de bază” cu rolul de a stoca caracteristici comune unor clase diferite, astfel aceste proprietăţi nu vor trebui precizate în fiecare clasă în parte.

Sau – permite extinderea unei clase existente si construirea unei noi solutii pe baza unei solutii existente fara a o modifica pe aceasta;

<u>Motive pentru a folosi mostenirea:</u>

- ✓ Refolosirea claselor(clasele folosite mai mult sunt si testete mai mult, deci mai fiabile).
- ✓ Standartizarea comportamentului unui grup de clase.
- ✓ Polimorfizmul (folosirea aceluiasi cod pentru clase diferite).

Avantajele:

- ✓ Putem crea noi clase care preiau caracteristicile unei clase deja definite;
- ✓ Scriem mai puțin cod prin reutilizarea parții comune celor două clase implicate în proces(cea din care se face derivarea și cea din care moștenește caracteristicile);
- ✓ Ne putem alege modul în care facem derivarea prin modificatorii de acces: public, private, protected;

Noțiuni:

- ✓ Clasa existenta, care va fi mostenita se mai numeste clasa de baza sau superclasa. Clasa care realizeaza extinderea se numeste subclasa, clasa derivata sau clasa descendenta.
- ✓ clasă poate mosteni proprietăți doar de la o singura clasa.
- ✓ In Java orice clasa este derivata/mostenita din clasa Object
- ✓ Subclasele mostenesc toate atributele si metodele neprivate ale claei de baza.
- ✓ Subclasele pot adauga atribute si metode.
- ✓ Subclasele pot redefini metode.
- ✓ Constructorii nu sunt mostiniti niciodata.
- ✓ In java o clasa nu poate extinde/mosteni mai multe clase.

Exemple:

IS-A relationship:

```
public class Animal{    }
```

```
public class Mammal extends Animal{   }
```

```
public class Reptile extends Animal{   }
```

```
public class Dog extends Mammal{   }
```

Analizând exemplu de mai sus, in terminologie POO, acesta inseamnă

că:

- ✓ Clasa Animal este superclasa clasei Mammal.
- ✓ Clasa Animal este superclasa clasei Reptile.
- ✓ Clasa Mammal si clasa Reptile sunt subclase a superclasei Animal.
- ✓ Dog este subclasa ambelor clase Mammal si Animal.
- ✓

Acum , bazândure pe relatia IS A, putem spune si vedea că:

- ✓ Mamal IS A Animal.
- ✓ Reptile IS A Animal.
- ✓ Dog IS A Mammal.

Cu utilizarea cuvintului rezervat de Java ”extends”, subclasele pot mosteni proprietăți de la superclase, cu exceptia proprietatilor private din superclasa.

HAS-A relationship:

Această relație ne ajută sa reducem dublicarea de cod:

```
public class Vehicle{}

public class Speed{}

public class Van extends Vehicle{

      private Speed sp;

}
```

Mostenirea multipla:

- ✓ O clasa poate mosteni:
  -o singura (!) clasa: **class** *Cat* **extends** *Animal*
  -una sau mai multe interfete: **class** *Cat* **implements** *Touchable, Jumping*
  -o singura clasa si una sau mai multe interfete: **class** *Cat* **extends**

Animal **implements** *Touchable, Jumping*

✓ Cand sunt mostenite mai multe interfete, acestea sunt enumerate dupa cuvantul cheie implements.

✓ ***Polimorfizmul*** este abilitatea unui obiec de a lua mai multe forme.

Sau – într-o ierarhie de clase obtinuta prin mostenire, o metodă poate avea implementari diferite la nivele diferite in acea ierarhie;

Sau – Este abilitatea de a procesa obiectele în mod diferit, în funcție de tipul sau de clasa lor. Mai exact, este abilitatea de a redefini metode pentru clasele derivate. De exemplu pentru o clasă Figura putem defini o metodă arie. Dacă Cerc, Dreptunghi, etc. ce vor extinde clasa Figura, acestea pot redefini metoda arie.

S*au*– permite implementarea de solutii diferite sub aceeasi denumire.

Sau - permite ca aceeasi operatie sa se realizeze in mod diferit in clase diferite. Sa consideram, de exemplu, ca in clasa *Figura_geometrica* exista metoda *arie()*, care calculeaza aria figurii respective. Clasele *Cerc, Triunghi, Patrat* sunt subclase ale clasei *Figuri_geometrice* si vor mosteni, deci, de la aceasta metoda *arie()*. Este insa evident ca aria cercului se calculeaza in alt mod decat aria patratului sau cea a triunghiului. Pentru fiecare din instantele acestor clase, la calcularea ariei se va aplica metoda specifica clasei sale.

Esenta polimorfismului este supraincarcarea si suprascrierea datelor.

Supraincarcarea:

- ✓ Doua metode din aceeasi clasa se numesc supraincarcate(overloaded) daca: au acelasi nume, difea prin numarul sau tipul argumentelor, nu difera doar prin tipul returnat.
- ✓ Numele unei metode impreuna cu numarul si tupul argumentelor se numeste semnatura metodei.
- ✓ Defineste metode noi(cu semnatura diferita) in aceeasi clasa.

Suprascrierea:

- ✓ Metoda din clasa parinte poate fi suprascrisa(overrided) de o metoda din clasa derivata daca: au acelasi nume, au acelasi numar si tipul de argumente, au acelasi tip returnat, drepturile de acces nu sunt mai restrinse.
- ✓ O metoda nu poate fi suprascrisa decat o data intr-o clasa.
- ✓ Suprascrierea implementeaza fiferit aceeasi metoda(cu a aceiasi semnatura) in diferite clase.
- ✓ Suprascrierea este esenta plimorfismului.

Noțiuni:

- ✓ Orice obiect din Java care poate avea mai mult de IS-A este considerat obiect polimorf.

Exemplu:

```
public interface Vegetarian{}
public class Animal{}
public class Deer extends Animal implements Vegetarian{}
```

, prin urmare clasa Deer este considerată a fi polimorfă fiindcă are mostenire moltiplă, deoarece:

- ✓ A Deer IS-A Animal
- ✓ A Deer IS-A Vegetarian
- ✓ A Deer IS-A Deer
- ✓ A Deer IS-A Object

✓ ***Încapsularea*** **-** este procesul in care are loc ascunderea implementarii unui obiect față de majoritatea clientilor sai.

Sau ne permite sa privim obiectul ca pe o "*cutie neagra*".

Sau – numită și ascunderea de informații: Asigură faptul că obiectele nu pot schimba starea internă a altor obiecte în mod direct (ci doar prin metode puse la dispoziție de obiectul respectiv); doar metodele proprii ale obiectului pot accesa starea acestuia. Fiecare tip de obiect expune o interfață pentru celelalte obiecte care specifică modul cum acele obiecte pot interacționa cu el.

Sau – contopirea datelor cu codul (metode de prelucrare si acces la date) în clase, ducând la o localizare mai bună a erorilor şi la modularizarea problemei de rezolvat;

Sau – metodologie prin care se ascunde cat mai mult din mecanismele interne ale clasei; atributele obiectelor nu sunt accesibile direct, ci doar prin intermediul interfetei (colectie de functii); in afara clasei se stie doar ce face clasa, insa nu se stie cum si nici nu ai posibilitatea de a modifica acest lucru;

Sau - este proprietatea obiectelor de a-si ascunde o parte din date si metode. Din exteriorul obiectului sunt accesibile ("vizibile") numai datele si metodele *publice*. Putem deci sa ne imaginam obiectul ca fiind format din doua straturi.

<u>Avantajele:</u>

- Separarea interfetei unui obiect de reprezentarea lui permite modificarea reprezentarii fara a afecta in vreun fel pe nici unul din clienti, intrucat acestia depind de interfata si nu de reprezentarea obiectului-server.
- Incapsularea permite modificarea programelor intr-o maniera eficienta, cu un efort limitat si bine localizat.
- Putem restrictiona sau putem permite accesul altor obiecte asupra atributelor sau metodelor unui obiect.

Exemplu:

Clasa angajat *(cuprinde o entitate din lumea reală cu toate atributele specifice ei).*
Atribute: *Nume; Prenume; CNP; Salariu; etc.*
Iar anumite ***Metode*** ar fi: calculează salariu, micsorează salariu, calculează zilele lucratoare și alte metode specifice salariului;
Și mai avem o ***Clasa clienți***, și dorim că acel client să știe numele, prenumele angajaților, doar poate nu dorim să știe și CPN-ul sau salariul lor.

Prin **Incapsulare** putem evita accesul lor la aceste doar prin modificatorii de acces ***private*** – nici un obiect nu are acces, ***public*** – orice alt obiect are acces, ***protected*** – doar obiectele derivate au acces, ***default*** – doar obiectele din acelasi pachet au acces.

Incapsularea datelor:

- ✓ Obiectele sunt ca o capsula. Structura de date este inconjurata de o interfata.
- ✓ Capsula defineste operatiile(procedurile, metodele) ce pot fi aplicate obiectului(apelate, activate).
- ✓ Structura de date poate fi ascunsa complect in interiorul obiectului.

- **Abstractizarea** inseamna eliminarea sau ascunderea deliberata a unor detalii ale unui proces sau artefact pentru a releva mai clar alte aspecte, detalii sau structura.

  Sau - posibilitatea ca un program să ignore unele aspecte ale informației pe care o manipulează, adică posibilitatea de a se concentra asupra esențialului. Fiecare obiect în sistem are rolul unui "actor" abstract, care poate executa acțiuni, își poate modifica și comunica starea și poate comunica cu alte obiecte din sistem fără a dezvălui cum au fost implementate acele facilitați. Procesele, funcțiile sau metodele pot fi de asemenea abstracte, și în acest caz sunt necesare o varietate de tehnici pentru a extinde abstractizarea;

  Sau - exprima toate caracteristicile esentiale ale unui obiect, care fac ca acesta sa se distinga de alte obiecte; abstractiunea ofera o definire precisa a granitelor conceptuale ale obiectului, din perspectiva unui privitor extern.

Notiuni:

- ✓ Metodele sunt doar declarate , pentru a putea fi apelate, dar implementarea lor se face in clasele derivate:
  abstract tip numeMetoda(*lista de argumente*);
- ✓ O clasa ce are cel putin o metoda abstracta trebuie sa fie abstracta: **abstract** *numeClasa.*
- ✓ Clasele abstracte nu pot fi instantiate!.
- ✓ Clasele abstracte trebuie sa implimenteze metodele abstracte mostenite sau sunt la randul lor abstracte.
- ✓ Clasele abstracte pot avea constructori. Acestia pot fi apelati din constructorii claselor derivate.
- ✓ O clasa abstracta poate avea metode implementate.
- ✓ O clasa abstracta poate avea variabile de instranta si de clasa.

- ✓ Acestea vor fi mostenite de clasele derivate si vor face parte(mai putin cele statice) din obiectele de tipul clasei derivate.
- ✓ Se poate declara abstracta o clasa chiar daca are toate metodele implementate. In felul acesta se interzice posibilitatea instantierii de obiecte de acel tip. Clasa poate fi insa folosita prin mostenire.
- ✓ Metodele abstracte obliga proiectantul de clase derivate sa le implementeze.
- ✓ Rolul lui final in mostenire este ca daca nu dorim suprascrierea unei metode, o declaram final. In clasele derivate nu se vor putea declara metode cu ceeasi semnatura.
- ✓ O metoda abstracta nu poate fi final pentru ca nu am putea sa o implementam.
- ✓ Daca nu dorim mostenirea unei clase, o declaram final.

Suprascrierea și supraîncarcarea

Suprascrierea apare atunci cind o subclasa declara o metoda care are aceleasi tipuri de argumente ca si metoda declarata in una din superclase.

Note:

- ✓ Suprascrierea nu poate avea mai multi modificatori de acces
- ✓ Supraâncarcarea se folosește pentru a exprima relația de tipul IS-A și se folosește pe plase de același fel(similare parțial).
- ✓ În practică înseamnă că un operator sau o funcție este definită de mai multe ori , diferența definirii fiind dată de tupul parametrului sau numărul lor.
- ✓ Supraâncarcarea exprimă forma de polimorfizm(efectuează diferite operații în forma de context).
- ✓ Funcţiile membre, moştenite de la clasa de bază, lucrul cărora nu

este satisfăcător pentru clasa derivată, trebuie să fie **suprascrise** în clasa derivată.

Totodată, funcţiile membre moştenite pot fi şi **supraîncărcate** în clasa derivată. Atât suprascrierea, cât şi supraîncărcarea funcţiilor membre în clasa derivată se face la fel ca şi definirea funcţiilor membre: în declararea clasei derivate descriem numai antetul funcţiei, iar definirea funcţiei o scriem după declararea clasei sau scriem toată definirea funcţiei în declararea clasei.

Trebuie de înţeles bine că funcţia care suprascrie sau supraîncarcă o funcţie membră a clasei de bază nu are, spre deosebire de predecesoarea sa, acces direct la membrii privaţi ai clasei de bază. În funcţia care suprascrie sau supraîncarcă o funcţie membră moştenită ultima poate fi apelată de mai multe ori, folosind **operatorul de rezoluţie ::** (se mai numeşte de scop), dar numai în cazul când ultima nu este privată în clasa de bază.

Exemplu:

```
class Animal{


  public void move(){
     System.out.println("Animals can move");
  }
}


class Dog extends Animal{


  public void move(){
     System.out.println("Dogs can walk and run");
  }
}


public class TestDog{


  public static void main(String args[]){
     Animal a = new Animal(); // Animal reference and object
     Animal b = new Dog(); // Animal reference but Dog object


     a.move();// runs the method in Animal class
```

```
    b.move();//Runs the method in Dog class
  }
}
```

Rezultat:

```
Animals can move
Dogs can walk and run
```

Modificatorii de Acces

Modificatorii de acces reprezinta modalitati prin care programatorul poate controla (acorda sau restrictioneaza) accesul la metodele si atributele definite intr-o clasa.

Există 4 tipuri de modificatori de acces: public, protected, default , private.

- **Public** înseamnă accesul de oriunde

  Sau: atributele și metodele sunt vizibile și pot fi accesate din exterior.

Notă:

✓ Un câmp declarat public poate fi schimbat de oriunde, fără nici o protecție.

✓ Metodele ar trebui să fie public numai dacă este de dorit să poată fi apelate din afara clasei.

Exemple:

Daca se considera clasa **ClassA** definita in pachetul **main**:

```
package main;

public class ClassA{
   public int atributPublic;       //public
}
```

atunci, atributul public este vizibil, sau poate fi accesat in **clase din acelasi pachet**:

```
package main;   //acelasi pachet

public class ClassB {
   public static void faCeva()
   {
       ClassA ref = new ClassA();
       ref.atributPublic = 20;
   }
}
```

atunci, atributul public este vizibil, sau poate fi accesat in **subclase ale clasei parinte din acelasi pachet**:

```
package main;   //acelasi pachet

public class SubClassA extends ClassA {
   public void faCeva(){

      ClassA ref = new ClassA();

      //accesibila prin referinta
      ref.atributPublic = 20;

      //accesibila prin mostenire
      this.atributPublic = 30;
   }
}
```

atunci, atributul public este vizibil, sau poate fi accesat in **subclase ale clasei parinte din alte pachete**:

```
package other;  //alt pachet

import main.ClassA;

public class OtherSubClassA extends ClassA {
   public void faCeva(){

       ClassA ref = new ClassA();

      //accesibila prin referinta
      ref.atributPublic = 20;

      //accesibila prin mostenire
      this.atributPublic = 30;}
```

atunci, atributul public este vizibil, sau poate fi accesat in **clase din alt pachet**:

```
package other;  //alt pachet

import main.ClassA;

public class ClassC {
   public static void faCeva()
   {
```

```
        ClassA ref = new ClassA();

        ref.atributPublic = 20;             //vizibila
    }
}
```

- **Protected** înseamnă accesibil din toate clasele din același director și accesibil din toate subclasele de oriunde.
  Sau: atributele și metodele sunt vizibile din clasa din care fac parte și din clasele derivate din ele;

Notă:

Exemple:

```
package main;

public class ClassA{

   protected int atributProtected;     //protected

   public void faCeva(){
        //vizibil in aceeasi clasa
        atributProtected = 20;
    }
}
```

Atributele protected sunt vizibile in:

- clase din acelasi pachet:

```
package main;    //acelasi pachet

public class ClassB {
   public static void faCeva(){
        ClassA ref = new ClassA();
        ref.atributProtected = 40;
    }
}
```

- subclase din acelasi pachet (prin mostenire si prin referinte)

```
package main;    //acelasi pachet

public class SubClassA extends ClassA {
   public void faCeva(){

       ClassA ref = new ClassA();

       //accesibil printr-o referinta
       ref.atributDefault = 20;

       //accesibil prin mostenire
       this.atributDefault = 30;
    }
}
```

- subclase din alte pachete (**IMPORTANT ! doar prin mostenire**):

```
package other;  //alt pachet

import main.ClassA;

public class OtherSubClassA extends ClassA {
   public void faCeva(){

        ClassA ref = new ClassA();

       //NU este accesibil prin utilizarea unei referinte
       ref.atributProtected = 20; //eroare de compilare

       //accesibil doar prin mostenire
       this.atributProtected = 30;
   }
}
```

Atributele si metodele protected NU sunt vizibile in clase din afara pachetului:

```
package other;  //alt pachet

import main.ClassA;

public class ClassC {
   public static void faCeva(){
        ClassA ref = new ClassA();
        //NU este vizibil - NU poate fi accesat
        ref.atributProtected = 20;  //eroare de compilare
   }
}
```

- **Private** înseamnă accesibil doar din această clasă.

  Sau: blochează accesul la atribut sau metoda din afară clasei parinte fără nici o excepție.

  Sau: atributele sau metodele pot fi accesate doar din metode aparținând clasei parinte.

Notă:

✓ Un câmp făcut private nu este ascuns față de obiectele de aceeasi clasă.

Exemple:

```
package main;

public class ClassA{
   private int atributPrivat;      //privat

   public void doSomething(){
       //vizibila doar in aceeasi clasa
       atributPrivat = 20;
   }
}
```

- **Default** înseamnă accesibil din toate clasele, subclasele din același director/packet.

Exemple:

Pentru clasa **ClassA**:

```
package main;

public class ClassA{

   int defaultAttribute;              //default

   public void faCeva(){
       //vizibil in aceeasi clasa
       atributDefault = 20;
   }
}
```

atributul ***atributDefault*** este vizibil in alte clase din acelasi pachet:

```
package main;    //acelasi pachet

public class ClassB {
   public static void faCeva(){
       ClassA ref = new ClassA();
       ref.atributDefault = 30;
   }
}
```

si in **subclase** din acelasi pachet:

```
package main;    //acelasi pachet

public class SubClassA extends ClassA {
   public void faCeva(){

      ClassA ref = new ClassA();

      //accesibil printr-o referinta
      ref.atributDefault = 20;
```

```
        //accesibil prin mostenire
        this.atributDefault = 30;
    }
}
```

Atributele si metodele default NU sunt vizibile in clase sau subclase din alt pachet:

```
package other;  //alt pachet

import main.ClassA;

public class ClassC {
    public static void faCeva(){
        ClassA ref = new ClassA();
        //NU este vizibil - NU poate fi accesat
        ref.atributDefault = 20;  //eroare de compilare
    }
}
```

Asa cum reiese din scurta descriere a modificatorilor de acces, acestia reprezinta reguli cu privire la dreptul de a accesa membrii (atribute si metode) unei clase din alte clase.

Sumarizarea regulile si scenariile de utilizare a modificatorilor de acces:

Vizibilitate (acces)	public	protected	private	default
Aceeasi clasa	X	X	X	X
Clase in acelasi pachet	X	X		X
Subclasa in acelasi pachet	X	X		X
Subclasa in alt pachet	X	X (doar prin mostenire)		
Clase in afara pachetului	X			

Aceast topic este important pentru testul SCJP deoarece daca nu se citeste cu atentie intrebarea exista riscul sa nu observi un modificator de acces de tip **private** sau **default** care va ascunde membrii unei clase (aribute sau metode) sau va genera eroare de compilare:

- o clasa are acces intotdeauna la propriile atribute si metode, indiferent de modificatorul de acces al acestora;
- membrii protected sunt vizibili in subclase din alte pachete DOAR

prin mostenire; asta inseamna ca pot fi accesati doar prin referinta this si nu printr-o referinta de tipul clasei parinte (vezi exemplele de la protected);

- inainte de a verifica modificatorii de acces ai membrilor (atribute sau metode) unei clase, verifica modificatorii de acces ai clasei (default or public daca clasa NU este vizibila, atunci nici membrii ei NU sunt, chiar daca sunt definiti public;

Tablouri de Obiecte

✓ Sunt grupuri de obiecte de acelasi fel. Exemplu: vector de numere; sir de caractere; grupa de studenti; teanc de CV-uri, uneme mai scurte, altele mai lungi.

✓ Obiectele dintr-un tablou de obiecte sunt ordonate si fiecare obiect are un numar de ordine, numit index. Accesul unui obiect din tablou se face pe baza acestui index.

Exemplu de sir de caractere:

L M M J V S D

Index 0 1 2 3 4 5 6

- Java permite construirea tablourilor multidimensionale neregulate.
- Obiectul tablou incapsuleaza elementele tabloului si un parametru care indica numarul de elemente, lungimea tabloului(lenght).
- Numele tabloului este o variabila de referinta la tabloul propiu-zis.

Declararea unui tablou de tip primitiv:

✓ Se specifica tipul elementelor si numele tabloului:

Tip [] numeTablou;

✓ Parantezele patrate indica faptul ca este vorba despre o declaratie de obiect tablou si nu o declaratie de variabila tip primitiv

Crearea tabloului de variabile de tip primitiv:

- ✓ Ca orice obiect din Java un tablou de variabile este creat(instantiat) cu ajutorul instructiunii **new**:
  numeTablou = **new** ***Tip*** *[nrElemente];*
- ✓ Specificarea numarului de elemente este obligatorie.
- ✓ Numele de elemente (lungimea tabloului) este un camp al obiectului tablou, numit lenght, care este initializat la instantierea obiectului tablou si care nu poate fi modificat.
- ✓ Campul lenght este accesibil ca orice camp public al unui obiect oarecare:
  numeTablou.lenght. Dar numai pentru citire.

Accesul la elementele tabloului:

- ✓ Dupa instantierea elementelor tabloului pot fi accesate folosind sintaxa clasica de acces pe baza indexului care indica pozitia elementului: *numeTablou[indexElement]*
- ✓ Indexul primului element este 0.
- ✓ *indexElement* este fie o constant intreaga pozitiva, fie o variabila de tip int, fie o expresie al carei rezultat este de tip int: *numeTablou[indexElement]*.

Exemple:

a[5]
a[i]
a[100 + i +j]
a[(int) (**Math.** Random()) * a.lenght]
,unde i si j sunt variabile de tip int, iar expresia double care trebuie convertit in tipul int(are loc si o trunchiere).

Exemplu un tablouu simplu de intregi:

```
int[] a; //declararea tabloului
a =new int[3];//crearea obiectului tablou
a[0] = 10; //accesul unui element al tabloului
a[3] = 9; // acces nepermis
```

Initializarea tabloului de tip primitv:

- ✓ Declararea tabloului poate fi combinata cu initializare folosind sintaxa:
  Tip[] numeTablou = {element0, element1, ...};
- ✓ Lungimea tabloului va fi fixata de numarul de termeni dintre acolade.
- ✓ Fiecare termen dintre acolade poate fi: constanta, expresie.
- ✓ Care are insa neaparat tipul corespunzator declaratiei tabloului.

Exemple:

```
//tablou de 10 intregi
Int[]  prim = {2,3,4,5,6,8,7, 8, 45,6};

// tablou de 7 caractere
char[] zi = {'L','M','M','J','V','S','D'};

//tablou de 2 intregi
int g1 = 421;
int g2 = 422;
int [] grupeExamen = {g1,g2 + 2};
```

Declararea unui tablou de obiecte:

- ✓ Se specifica tipul elementelor si numele tabloului:

NumeClasa[] numeTablou;

- ✓ Elementele tabloului sunt referinte la obiectele de tipul clasei declarate.
- ✓ Tabloul de obiecte este de fapt un tablou de variabile referinta la obiect.

Exemplu:

- ✓ *Student [] grupaCIB103;*
- ✓ *Student [] gascaDeRevelion;*
- ✓ *String [] numeZi; // intre parantezele patrate nu se scrie nimic*

Crearea tabloului de obiecte:

- ✓ Ca orice alt obiect din Java un tablou de obiecte este creat (instantiat) cu ajutorul instructiunii new:
  numeTablou = new NumeClasa[nrElemente]; // specificarea numarului de elemente este obligatoriu.
- ✓ Numarul de elemente(lungimea tabloului)este un camp al obiectului tablou, numit lenght, care este initializat la instantierea obiectului tablou si care nu poate fi modificat.
- ✓ Campul lenght este accesibil ca orice camp public al unui obiect oarecare:
  numeTablou.lenght // dar numai pentru citire.
- ✓ La instantierea unui tablou de obiecte, elementele tabloului, variabile referinta la obiect, sunt initializate in null.
- ✓ Dupa instantiere tabloului, elementele lui nu pot fi accesate decat dupa ce referitelor li se atribuie o adresa valida, fie prin instantierea unui obiect corespunzatot, fie prin copierea unei referinte la un obiect de tipul declarat.

Accesul unui element al unui tablou de obiecte:

- ✓ Accesul unui element al unui tablou de obiecte este accesul la obiectul la care face referire elementul proupriu-zis al tabloului. Obiectul este identificat prin numele tabloului si indexut la care se afla:
  numeTablou[indexElement].
- ✓ Indexul are acelasi constringeri ca in cazul tablourilor de variabile de tip primitiv.
- ✓ Accesul membrilor(campurisau metode) obiectului se face folosind sintaxa clasica de acces a obiectelor:
  numeTablou[indexElement].numeCamp sau
  numeTablou[indexElement].numeMetoda.

Exemplu: un tablou de obiecte:

Student[] *grupa; //declararea tabloului*
grupa = new ***Student[3];*** *//crearea tabloului*
grupa[0] = new ***Student();*** *// instantierea unui obiect Student care face parte din tablou.*
grupa[1] =new ***Student();*** *//instantierea altui obiect Student care face parte din tablou.*
grupa[1].nr_credite =60; //accesul unui element

Initializarea tablourilor de obiecte:

- ✓ Declararea unui tablou de obiecte poate fi combinata cu o initializare folosind sintaxa cu paranteze acolade:
  NumeClasa [] numeTablou = {elemetn0, element1,...};
- ✓ Lungimea tabloului va fi fixata de numarul de termeni dintre acolade.
- ✓ Fiecare termen dintre acolade poate fi: referinta, instantiere.
- ✓ Care are insa neaparat clasa corespunzatoare declaratiei tabloului.

Exemplu de initializare prin referinte:

Student s_a = new **Student**();
Student s_b = new **Student**();
Student s_c = new **Student**();
Student [] grupProiect = {s_a, s_b, s_c};

Exemplu de initializare prin instantiere:

***Student** [] grupProiect = {new **Student**(), new **Student**(), new **Student**()};*

Tablouri bidimensionale:

- ✓ Tablourile fiind obiecte pot fi la randul lor grupate in tablouri de obiecte.
- ✓ Tabloul bidimensional este un tablou de obiecte tablou.
- ✓ Tabloul de pe primul nivel este un tablou de referinta la tablourile de pe al doilea nivel.
- ✓ Tablourile de pe ultimul nivel pot fi tablouri de variabile de tip primitiv sau tablouri de obiecte (de referinte la obiecte).
- ✓ Taoate tablourile de pe ultimul nivel au elemente de acelasi tip, tipul din declaratia tabloului bidimensional.
- ✓ Tablourile de pe ultimul nivel pot fi de lungimi diferite.

Constructori și Destructori

- **Constructorul** este sigura metoda pentru a crea instanțe ale unei clase.

  Sau: este responsabil de a crea obiecte.

Constructor:

Scop	Inițializează starea unui obiect
Nume	La fel cu numele clasei. Prima literă mare, stil "cămilă"

Cod	`public Taxi(){`
	`  ...`
	`}`

Ieșire	Nu este tip de retur în antet
Intrare	0 sau mai mulți parametri
Utilizare	`> Taxi cab;` `> cab = new Taxi();`
# de apelări	Cel mult o dată pe obiect; invocat de operatorul "new"

Constructori multipli

```
public class Taxi{
 private int km;
 private String driver;

 public Taxi(){
    km = 0;
    driver = "Unknown";
 }

 public Taxi(int km,String
d){
    this.km = km;
    driver = d;
 }
```

O operație "new" încununată de succes creează un obiect pe heap și execută constructorul al cărui lista de parametri "corespunde" listei sale de argumente (ca număr, tip, ordine).

```
> Taxi cab1;
> cab1 = new Taxi();
```

```
> Taxi cab2;
> cab2 = new
Taxi(10,"Jim");
```

Note:

- ✓ Au același nume cu clasa și nu au tip returnat.
- ✓ Sunt apelați automat de către compilator;
- ✓ Practic se cere ca ei să fie definiți în domeniul de vizibilitate public.
- ✓ Fiecare clasa are cel putin cite un constructor, daca nul cream noi, la compilarea programului java il creaza implicit.
- ✓ Constructorii se folosesc pentru initializarea datelor de obiect.
- ✓ Construcotrii sunt supraincarcati(overloaded - mai multi construcori cu acelasi nume).
- ✓ Permite initializarea personalizate a obiectelor.
- ✓ Constructorii se folosesc numai la crearea unui obiect.(operatorul de instantiere new si constructorul Student(); si in rezultat avem ***new Student();***).
- ✓ Instantiere unui obiect presupune construire lui.
- ✓ Operatourl de instantiere , new, aloca memorie pentru obiect si initializeaza toate cimpurile obiectului cu valorile implicite si , daca e cazul, cu cele specifice in declararia campurilor si blocurilor de initializare.
- ✓ Constourctorul are si rol de a declara la instantiere clasa obiectului ce urmeaza a fi creat, deoarece constructorul are numele identic cu al clasei pe care il construieste.
- ✓ Constructorii nu pot fi mosteniti, desi clasele derivate pot apela constructorul clasei de baza(superclasa).

Avantajele:

- ✓ Prezența zonelor de vizibilitate face imposibilă inițializarea corespunzătoare a obiectelor din afara clasei.
- ✓ Nu e nevoie de intervenția programatorului pentru a apela metode de inițializare.
- ✓ Obiectele nu pot fi inițializate direct ca structurile tocmai datorită zonelor de vizbilitate.
- ✓ Cînd nu sunt definiți de programator, cel implicit este creat automat de compilator.

In orice metoda de instanta(nestatica):

this - este o referinta la obiectul curent.(This este folosit pentru a face diferenta intre atrubutele obiectului si parametri cu acelasi nume dintr-o metoda de instanta, sau pentru a apela un alt constructor.)

super - este o referinta la obiectul parinte.(Super este folosit pentru a deosebi metodele cu acelasi nume din subclasa si din suparclasa si pentru a apela construcotrii clasei de baza).

Intr-o clasa se pot defini metode, dar exista un tip special de metode care sunt folosite pentru a rezolva o anumita problema, aceea de a construi obiecte. Constructori sunt metode speciale datorita rolului lor si pentru ca au o multime reguli privind declararea si utilizare.

De fiecare data cand este creat un obiect, este apelat un constructor. Pornind de la aceasta certitudine, în Java, **fiecare clasa are cel putin un constructor** , chiar daca programatorul nu a declarat în mod explicit unul.

Rolurile functiei constructor sunt:

- **Rolul principal** – pentru a construi obiecte, in sensul de a aloca spatiu în Heap;
- **Rol secundar [optional]** – pentru a initializa variabilele de instanta (atribute) cu valori default (amintiti-va, ca variabilele de instanta primesc valoarea implicita a tipului lor atunci cand obiectul este creat) sau cu valori date;

Exemplu:

Avand în vedere clasa **Carte** si metoda **main():**

```
public class Carte {
    float pret;
    String titlu;
    String autor;

    public static void main(String[] args)
    {
        //construieste un obiect de tip Carte
        Carte c1 = new Carte();
    }
}
```

este clar ca in metoda **main()** este construit un obiect de tip **Carte**, ce este gestionat de referinta ***c1***.

In aceasta situatie, unde este constructorul ? O regula în ceea ce priveste constructorii afirma ca, compilatorul va oferi un constructor implicit (unul cu zero argumente) **daca nu exista declarati in mod explicit alti constructori.** Forma constructorului implicit, generat de compilator, este:

```
    public Carte()
    {

    }
```

Reguli pentru a declara si apela constructori in Java:

- ✓ constructorii au acelasi nume (case-sensitive), ca si clasa parinte;
- ✓ constructori nu au un tip de return (este logic, deoarece vor intoarce intotdeauna o referinta catre obiectul construit); **ATENTIE** pot fi definite metode cu acelasi nume ca si clasa, dar cu un tip de return care sunt metode comune si NU constructori.
- ✓ Poate avea argumente.
- ✓ Poate ave modificatori de acces.
- ✓ Nu poate ave alti modificatori(static, final, etc.)

Exemle:

*~~final~~ public **Student** (String str, int nr){*
*//*corpul constructorului*
}

Implementarea constructorului:

- ✓ Nicaieri nu apare return in corpul constructorului

- ✓ Poae folosi orice camp de tip primar al clasei.
- ✓ Poate folosi campurile de tip referinta numai daca au fost initializate anterior.

Supraincarcarea constrcutotrilor(ovrloading):

- ✓ Un constructor este supraincarcat daca sunt definite mai multe varinte.
- ✓ Mai mult constructori = mai multe posibilitati de instantiere.
- ✓ O clasa poate avea oricati constructori.
- ✓ Tori constructorii au acelasi nume.
- ✓ Variantele constructorului difera prin numarul si/sau tipul argumentelor.
- ✓ Alegerea constructorului la instantiere se face dupa semnatura.

Exemple:

Student *st1, st2;*
st1 = new ***Student****(60, "Popescu");*
st2 = new ***Student****();*

- ✓ In primul caz JVM(Java Virtual Machine) ca alege constructorul cu doua argumente, in al doilea, constructorul fara argumente.
- ✓ Daca o clasa are (Cel putin) un constructor cu doua argumente, ca in acest exemplu, instantierea fara argumente este permisa numai daca este implementat constructorul constructorul fara argumente. Compilatorul va semnala eroare daca nu-l gaseste.

Inlantuire constructorului:

- ✓ La instantierea unui obiect un singur constructor este atasat

operatorului *new*. Totousi, un constructor poate apela alt constructor.

- ✓ Mai multi constructori pot contine aceeasi secventa de instructiuni. Are sens atuci reutilizarea codului unui constructor de catre un alt constructor.
- ✓ Constructorii mai generali pot apela constructori cum multe argumente.
- ✓ Nu este permis apelul unui constructor din corpul aceluiasi constructor. Regresie infinite.
- ✓ Nu este permis apelul constructorului apelant din corpul apelatului. Bucle infinite.
- ✓ Nu este permins apelul unui constructor din corpul aceluiasi construcotri(regresie infinita).
- ✓ Nu este permis apelul constructorului apelant din corpul apelatului(bucle infinite).
- ✓ Un constructor poate apela un singur constructor.
- ✓ Permite reutilizarea si compactarea codului.
- ✓ Foloseste apelul unui constructor prin this(). This aceasta, acesta clasa.(this in combinatie cu o lista de argumente(care poate fi vida) reprezinta un apel de constructor din aceeasi clasa).
- ✓ Este apelat constructorul ai carui paramentri coincid cu tipurile argumentelor din lista atasata de this.
- ✓ Instructiunea this() apeleaza constructorul fara argumente, daca este implementat bineinteles. In nici un caz nu se poate apela constructorul implicit(Implementarea oricarui constructor duce la anularea constructorului implicit).
- ✓ Intrucit apelul unui constructor este obligatoriu prima instructiune din corpul constructorului epelant, constructorul apelat va fi executat inaintea constructorului apelant.
- ✓ Astfel spus initializarile constructorului apela sunt executate inaintea oricarei initializari din constructorul epelant.

<u>Instantierea: Ordinea initializarilor:</u>

- ✓ Alocarea spatiului de memorie pentru campurile membre.
- ✓ Initializari implicite.
- ✓ Initializari din declaratii.
- ✓ Blocuri de initializare.
- ✓ Executia instructiunilor din corpul constructorului.

Exemplu:

```
Class Student {
Int nr_credite = 60;
String nume;
{nume = "Popescu";}
Student (int nr){
nr_credite = nr_credite + nr;
}
}
```

- **Destructorul** responsabil de ştergerea obiectelor.

Note:

- ✓ Opus constructorului ca rol, el șterge obiectele din locul în care au fost definiți.
- ✓ Denumirea sa este dată de numele clasei.
- ✓ Spre deosebire de contructor, el apare o singură dată și nu are niciodată parametri de apel.
- ✓ Este pus automat(de compilator dacă nu este definit de utilizator așa cum se întimplă in majoritatea cazurilor).

Tipuri de Date

In Java exista 2 tipuri de variabile:

- ***Date primitive***
- ***Obiecte sau Referinte***

Tipurile primitive sunt tipuri de date fundamentale ce nu mai pot fi descompuse in alte subtipuri. In Java 6 exista 8 tipuri de date primitive:

Tip data	Dimensiune	Valori cu semn	Tip
byte	1 byte	-128 –> 127	intreg
short	2 bytes	-32768 –> 32767	intreg
int	4 bytes	-2147483648 –> 2147483647	intreg
long	8 bytes	-9,223,372,036,854,775,808 –> 9,223,372,036,854,775,807	intreg
float	4 bytes	7 cifre semnificative	real simpla precizie
double	8 bytes	15 cifre semnificative	real dubla precizie
char	2 bytes	'\u0000' –> '\uffff' 0 –> 65535	caracter Unicode pe 16 biti
boolean	1 bit	true sau false	logic

Pentru a defini o variabila se foloseste sintaxa:

tip_variabila nume_variabila;

unde:

tip_variabila – unul din cele 8 tipuri primitive sau un tip definit de programator prin clase;
nume_variabial – numele variabilei definit de programator;

Regulile de care se tine cont la definirea variabilelor:

- ✓ numele variabilei trebuie sa inceapa cu o litera, linie de subliniere (_) sau cu simbolul dolar ($);
- ✓ numele de variabila NU poate incepe cu o cifra;
- ✓ dupa primul caracter se pot folosi cifre
- ✓ numele de variabila NU poate fi un cuvant Java rezervat (Java

keywords);

- ✓ pot fi definite mai multe variabile in acelasi timp;

```
public class Variables {
   public static void main()
    {
        //variabile definite corect
        int vb1,vb2;
        float fvb2;
        double _temp;
        boolean $flag;

        //variabile definite gresit - eroare compilare
        byte 3vb;       // incepe cu o cifra
        long for;       //foloseste un cuvant cheie
    }
}
```

- ✓ numele de variabile sunt alese de catre programator, insa pentru eficienta exista o serie de conventii cu privire la numele variabilelor: notatia Ungara, CamelCase; desi nu este obligatorie, in Java exista o recomandare cu privire la numele variabilelor; aceasta este derivata dinCamelCase si presupune ca numele de variabile sa fie cat mai sugestive, iar daca sunt formate din mai multe cuvinte, doar primul cuvant se scrie cu litera mica;

```
int iNumarCarti;        //notatie Ungara
int NumarCarti;         //CamelCase
int numarCarti;         //Java mixed case
```

Reguli pentru initializarea variabilelor:

La initializarea unei variabile trebuie sa se tina seama de tipul acesteia, deoarece in Java NU este posibil atribuirea de valori de tip diferit decat cel al variabilei. De exemplu, instructiunea urmatoare genereaza eroare de compilare de tipul possible loss of precision:

```
float vb2 = 23.5;           //eroare compilare - possible loss of precision
int vb3 = 45.6;             //eroare compilare - possible loss of precision
boolean test = 23;          //eroare compilare  - incompatible types
```

In cazul variabilei de tip *float*, vb2, eroare este generata deoarece valorile reale constante sunt considerate de tip *double*. Corect este sa pui **f** la sfarsitul valorii, adica 23.5f.

- ✓ tipul valorii trebuie sa fie identic cu tipul variabilei;
- ✓ pot fi initializate mai multe variabile in acelasi timp;

- ✓ in Java singurele valori posibile pentru variabile boolene sunt true sau false (in C sau C++, orice valoare numerica diferita de 0 este considerata true);
- ✓ valorile constante de tip float se definesc cu simbolul f la final deoarece, implicit, constantele reale sunt considerate de tip double;
- ✓ simbolurile de tip caracter se definesc intre ‘ ’ (apostrof) si nu intre “ “ (ghilimele);
- ✓ valorile reale pot fi definite si in format stiintific; de exemplu, 123.4 este echivalent cu 1.234e2;
- ✓ valorile intregi in baza 8 sunt prefixate cu 0; de exemplu 021 este 17 in baza 10;
- ✓ valorile intregi in baza 16 sunt prefixate cu 0x; de exemplu 0×11 este 17 in baza 10;
- ✓ variabilele de tip char pot avea ca valori o serie de caractere speciale:

Caracter special	Valoare
\b	backspace
\t	tab
\n	line feed
\f	form feed
\r	carriage return
\”	ghilimele
\’	apostrof
\\	backslash

Exemple de initializari corecte de variabile cu tipuri de date primitive:

```
int value1;
int value2;
value1 = value2 = 17;
int  valueB8 = 021;
int valueB16 = 0x11;
float value3 = 123.4f;
double value4 = 123.4;
char c = 'a';
char enter = '\r';
boolean isNumber = true;
long value5 = 17L;
```

Valori implicite pentru variabile in Java:

Daca variabilele nu sunt initializate, atunci acestea iau valori implicite (NU in toate situatiile !):

Tip de data primitiv	Valoare default	Valori cu semn	Tip
byte	0	-128 –> 127	intreg
short	0	-32768 –> 32767	intreg
int	0	-2147483648 –> 2147483647	intreg
long	0L	-9,223,372,036,854,775,808 –> 9,223,372,036,854,775,807	intreg
float	0.0f	7 cifre semnificative	real simpla precizie
double	0.0d	15 cifre semnificative	real dubla precizie
char	‘\u0000’	‘\u0000’ –> ‘\uffff” 0 –> 65535	caracter Unicode pe 16 biti
boolean	false	true sau false	logic

Important ! Variabilele locale (definite in interiorul unei metode) NU sunt initializate cu valori implicite de catre compilator. Valorile implicite din tabelul anterior sunt folosite pentru atributele obiectelor (valori definite la nivelul clasei). Utilizarea unei variabile locale neinitializata va genera eroare de compilare:

```
public static void main(){

  int sum;              //variabila locala definita in metoda main
  sum = sum + 10;      //eroare compilare
                       //variable sum might not have been initialized
}
```

Tipuri intregi:

NUME TIP	DIMENSIUNE IN BYTES	DOMENIU
unsigned char	1	0..255
Char	1	-128..127
unsigned int	2	0..65535
short int	2	-32768..32767
Int	2	-32768..32767
unsigned long	4	0..4294967295
Long	4	-2147483648..2147483647

Tipuri reale:

NUME TIP	DIMENSIUNE IN BYTES	MAXIMA (in valoare absoluta)
Float	4	3.4*pow(10,38)
Double	8	1.7*pow(10.308)
Long double	10	1.1*pow(10,4932)

Instrucţiuni

In acest post vom vedea cum se implementeaza in Java, structurile de control fundamentale:

- structuri decizionale***: if – then, if – then – else,switch***
- structuri de ciclare ***for, do-while,while – do, for imbunatatit (enhanced – for)***
- instructiuni de control: ***break, continue;***

Prin intermediul structurilor de control putem scrie programe a caror executie nu inseamna doar o secventa liniara de instructiuni.

IF – aceasta instructiune specifica ce bloc de cod sa se execute in functie de rezultatul evaluarii unei conditii numite expresie boolene.

***Cum se implementeaza in Java* if – then:**

Structura conditionala if – then are forma:

```
1: if (conditie)
2:    < expresie 1 >
```

sau

```
1: if (conditie) {
2:    < expresie 1 >
3:    < expresie 2 >
4: }
```

in care (**ATENTIE !**)***conditie*** reprezinta o expresie sau variabila booleana ce are ca valoare *true* sau*false*. De exemplu *30 > 10* sau 30 == 10.

In Java nu se accepta conditii bazate pe expresii sau variabile care au valori numerice. De exemplu, expresia urmatoare este valida in C/C++ insa NU si in Java:

```
int valoare = 10;
if(valoare)
   System.out.println("Valoare nenula !");
```

genereaza eroare de compilare de tipul **incompatible types**.

Pentru a exemplifica sa determinam daca un numar este negativ, si daca este atunci sa il modificam:

```
int negativeNumber = -22;
boolean isNumber = true;

if(isNumber)
   System.out.println("Este numar !");

if(negativeNumber < 0)
{
   System.out.println("Este numar negativ ! Il vom face pozitiv;");
   negativeNumber = (-1) *negativeNumber;
   System.out.println(negativeNumber);
}
```

***Cum se implementeaza in Java* if – then – else:**

Structura conditionala if – then – else are forma:

```
1: if (conditie){
2:    < expresie 1 >
3:    < expresie 2 >
4: }
5: else{
6:    < expresie 3 >
7:    < expresie 4 >
8: }
```

Daca vrem sa determinam minimul dintre 2 numere:

```
int vb1 = 10;
int vb2 = 20;
int min;
if(vb1 < vb2)
   min = vb1;
else
   min = vb2;
```

Aceeasi solutie poate fi implementata si prin intermediul operatorului conditional ?::

conditie ? < expresie then > : < expresie else >

, iar exemplul de mai devreme devine:

```
min = vb1 < vb2 ? vb1 : vb2;
```

***Cum se implementeaza in Java* do – while:**

Structura repetitiva *do – while* implementeaza un ciclu post-conditionat, deoarece conditia care asigura iesirea/ramanerea in bucla se verifica la sfarsit. Structura va executa cel putin o data iteratia:

```
1: do
2: {
3:    < expresie 1 >
4:    < expresie 2 >
5: } while (conditie);
```

De exemplu, daca dorim sa calculam N! (cu conditia ca N > 1) printr-un *do – while*:

```
//n factorial prin do - while cu conditia ca n > 0
int nFactorial = 1;
int i = 0;

    int n = 5;
do
```

```
{
    i++;
   nFactorial = nFactorial *i;
}while(i < n);
//afisam valoarea
System.out.println(n+"! = "+nFactorial);
```

Cum se implementeaza in Java *while – do*

Structura repetitiva *while – do* implementeaza un ciclu pre-conditionat, deoarece conditia care asigura iesirea/ramanerea in bucla se verifica la inceput inaine de a se executa iteratia:

```
1: while (conditie)
2: {
3:    < expresie 1 >
4:    < expresie 2 >
5: }
```

Acelasi exemplu, N!, dar prin *while – do*:

```
//n factorial prin while - do
int nFactorial = 1;
int i = 0;

    int n = 5;
while (i < 5)
{
    i++;
   nFactorial = nFactorial *i;
}
//afisam valoarea
System.out.println(n+"! = "+nFactorial);
```

***Cum se implementeaza in Java* for:**

Structura repetitiva *for* implementeaza o structura repetitiva pre-conditionata, asemenea lui *while-do*. Structura *for* este mult mai eficienta doarece iteratia si initializarea sunt incluse in structura:

```
1: for(initializare; conditie; iteratie)
2: {
3:    < expresie 1 >
4:    < expresie 2 >
5: }
```

De exemplu pentru a determina suma elementelor unui vector:

```
int[] vector = {1,2,3,4,5};
int suma = 0;
for(int j = 0; j < vector.length; j++)
   suma += vector[j];
System.out.println("Suma este "+suma);
```

In interiorul structurii for, pot fi trecute mai multe instructiuni de initializare sau de iteratie separate prin , (virgula):

```
1: for(initializare1, initializare2; conditie; iteratie1, iteratie2)
2: {
3:    < expresie 1 >
4:    < expresie 2 >
5: }
```

Elementele instructiunii for sunt optionale. Urmatoarele exemple sunt corecte, insa in unele dintre ele trebuie sa decizi cand se termina bucla infinita prin **break**.

```
for( initializare; ; )
for( ; conditie; iteratie)
for( ; ; iteratie)
for( ; ; ) // bucla infinita
```

Pentru examenul SCJP, trebuie avut in vedere faptul ca variabilele declarate in zona de initializare reprezinta variabile locale blocului *for* si nu sunt vizibile in afara lui.:

```
for ( int i=0; i<10 ; i++ ) {
// prelucrari
}
// eroare compilare: cannot resolve symbol: i
System.out.println("valoarea lui i este " + i );
```

***Cum se implementeaza in Java* enhanced – for:**

Structura repetitiva *enhanced – for* implementeaza o structura repetitiva pre-conditionata. Aceasta structura a fost introdusa incepand cu Java 5.0 pentru a permite o sintaxa mai usoara (este echivalent cu *foreach* din .NET). Aceasta structura poate fi utilizata doar pentru a itera prin colectii care implementeaza interfata java.lang.Iterable

```
1: for ( variabila : colectie_iterabila )
2: {
3:       < expresie 1 >
4:       < expresie 2 >
5: }
6:
```

Suma elementelor unui vector cu enhanced – for arata asa:

```
suma = 0;
for(int valoare : vector)
{
   suma += valoare;
}
```

```
System.out.println("Suma este "+suma);
```

***Cum se implementeaza in Java* switch**

Structura conditionala *switch* implementeaza o structura conditionala cu mai multe ramuri de executie. Inlocuieste intr-un mod mai eficient o structura *if-then-else* cu multe ramuri *else* sau*then*.

```
switch (variabila) {
case valoare_constanta1:
    < expresie 1 >
    break;
case valoare_constanta2:
    < expresie 2 >
    break;
…
default:
     < expresie >
}
```

Este important ca fiecare expresie de tip *case* sa fie terminata cu instructiunea *break* deoarece aceasta asigura iesirea din structura.
De exemplu testarea valorii unei variabile se poate face mai usor prin *switch* decat prin mai multe structuri *if* imbricate.

```
int valoareTest = 2;
switch(valoareTest)
{
   case 1:
      System.out.println("Valoarea este egala cu 1");
      break;
    case 2:
      System.out.println("Valoarea este egala cu 2");
      break;
    case 3:
      System.out.println("Valoarea este egala cu 3");
      break;
     case 4:
      System.out.println("Valoarea este egala cu 4");
      break;
   default:
      System.out.println("Valoarea este in afara intervalului");
}
```

Prin executia exemplului anterior se obtine mesajul: **Valoarea este egala cu 2.**

Daca nu se foloseau instructiuni de tip break atunci exemplul anterior ar fi afisat:

Valoarea este egala cu 2
Valoarea este egala cu 3
Valoarea este egala cu 4
Valoarea este in afara intervalului

Cum se implementeaza in Java instructiunile break si continue:

Instructiunea *break* permite intreruperea unei bucle *for, do-while*, *while-do* sau iesirea dintr-o serie de *case*-uri.
Instructiunea *continue* permite trecerea la urmatoare iteratie a unui ciclu *for, do-while*, *while-do*ignorand restul instructiunilor din iteratia curenta.

De exemplu, fiind dat un vector sa se determine suma elementelor pozitive:

```
int[] valoriInt = {10,12,5,-4,3,-1,23};
int sumaPozitive = 0;
for(int j = 0; j < valoriInt.length; j++)
{
   if(valoriInt[j] < 0)   //daca este negativ
      continue;         //trecem la urmatoarea iteratie

   //in mod normal, adunam valoarea
   sumaPozitive+=valoriInt[j];
}
System.out.println("Suma elementelor pozitive este "+sumaPozitive);
```

sau sa se determine prima valoare negativa:

```
int[] valoriInt = {10,12,5,-4,3,-1,23};
int valoareNegativa = 0;
for(int j = 0; j < valoriInt.length; j++)
{
   if(valoriInt[j] < 0)
   {
      //daca este negativ
      //salvez valoarea
      valoareNegativa = valoriInt[j];
      //iesim din bucla
      break;
   }
}
System.out.println("Prima valoare negativa este  "+valoareNegativa);
```

Pointeri

***Pointerul* este un tip de dată predefinit, care are ca valoare adresa unei zone de memorie. Putem accesa o dată necesară dacă ii cunoaștem adresa.**

Exemplu:

Un poştaş aruncă in fiecare dimineață ziare în diverse curţi ce au abonament la o anumită publicație.
Poștașul nu poate ști direct dacă un abonat a citit sau nu publicația din acea zi.
Dar știind adresa la care a lăsat-o poate merge și intreba abonatul dacă a facut acest lucru sau nu.
Similar se manifestă și pointerul.

Declararea pointerilor:

Pointerii se declară în felul următor:

*tip_data *identificator;*

tip -_data este tipul de bază de care vorbeam (adică, tipul datei stocate la adresa memorată de pointer).
* - este operatorul de referinţă.

declarari de pointeri:
*int * number;*
*char * character;*
*float * greatnumber;*

Declararea variabilelor de tip referinta:

Tipul referinţă permite folosirea mai multor identificatori pentru aceeaşi variabilă (de memorie).

Declararea se face în felul următor:

tip_data &nume_var = nume_var_referita;

& este operatorul de *referenţiere* (adresare).

nume_var este numele variabilei care se defineşte prin referinţa la o altă variabila, în acest caz fiind vorba de variabila nume_var_referita.

Notiuni:

- ✓ In java nu exista adrese absolute (pointeri) doar referinte.
- ✓ Adresa obiectului este o valoare ce poate fi atrbuita dinami unor variabile referinta(compatibila ca tip)

- ✓ Valoare unei referinte nu se confunda cu obiectul;
- ✓ Declararea unei variabile referinta este sinonima cu crearea variabilei referinta(la fel ca in cazul variabilor de tip primitv).
- ✓ Declararea unei variabile referinta nu implica crearea niciunui obiect. O variabila referinta nou creata nu are adresa valida, nu face referire la nici un obiect. Valoarea ei initiala este null.
- ✓ Variabilile referinta au un tip, o clasa. Clasa variabilei referinta denota clasa obiectelor la care se poate face referire(a caror adrese pot fi stocate).
- ✓ Valoarea unei variabile referinta nu poate fi modificate decit prin atribuirea valorilor returnate la crearea unui nou obiect(adresa obiectului nou creat): contul_meu = new Cont(); sau prin atribuirea valorilor altei variabile de referinta de aceiasi clasa sau de un tip compatibil (clasa inrudita): contul_meu = contul_tau;
- ✓ Variabilile ca entitati adresabile direct din program, care pot fi desemnate prin ume, sunt de doua feluri: variabile primitive si variabile referinta.
- ✓ Obiectul nu poate fi accesat decit indirect, prin intermediul variabilelor referinta.
- ✓ Referintele de acelasi tip pot fi grupate intr-o singura declaratie, precum in primul exemplu de mai sus.

- ✓ O variabila referinta la obiect poate fi folosita pentru a accesa doar obiecte din clasa corespunzatoare.
- ✓ Pentru a accesa obiectul de tipul *Student* avem nevoie de o variabila de referinta de tip *Student*.
- ✓ Declararea unei referinte se face dand numele clasei dorit urmat de numele ales pentru referinta.

Exemple de referinta:

Sintaxa: *NumeClasa numeReferinta:*

Student student_1, student_2;
student_1 new Student();
student_2 new Student(); *// se creaza un obiect sudent si se ataseaza o referinta*

Alt exemplu:

- ✓ class Cont{
- ✓ int cont_curent;
- ✓ int depozit;}
- ✓ Contul contul_meu;
- ✓ Contul_meu = new Cont();

Enumerari

Enumerarile sunt de fapt clase in care se pot declara metode si atribute.

Exemplu:

Exista situatii in care o variabila trebuie sa aiba valori limitate la o anumita multime, definita in specificatiile solutiei. Sa presupunem ca trebuie dezvoltata o aplicatie Java care gestioneaza Vehicule iar tipul de motor trebuie sa ia o valoare din multimea {BENZINA, DIESEL, HYBRID, ELECTRIC}. Pentru a implementa cerinta se poate defini atributul asociat tipului de motor ca **String** sau ca **int** si se valideaza de fiecare data valoarea de intrare. Pentru siruri de caractere se poate compara valoarea de intrare cu "BENZINA", "DIESEL", si asa mai departe. Pentru **int** se poate face asocierea BENZINA este 1, DIESEL este 2, … si se verifica valorile pe baza acesti abordari. Aceasta este o solutie posibila, dar nu e eficienta pentru ca se pot face cu usurinta greseli si pentru ca se complica o procedura care ar trebui sa fie simpla.

Sintexa pentru a declara o enumerare este:

enum **nume_enumerare** {constanta1, constanta2, …, constantaN } ; **// ATENTIE ; (punctul si virgula) de la final este optional**

Enumerarile pot fi declarate:

- independent, la nivel global asemenea unei clase;
- intr-o alta clasa;
- **NU** in metode;

Revenind la scenariul cu vehicule, solutia optima este de a declara o **enumerare** pentru tipul de motor:

```
//enumerare definita independent asemenea unei clase
//ATENTIE ! ; de la final este optionala
enum TipMotor {DIESEL,BENZINA,HYBRID,ELECTRIC}

class Vehicul
{
  //enumerare definita in interiorul unei clase
  protected enum TipCulori{RED, GREEN, BLUE, WHITE};

  //atribut instanta  de tip TipMotor
  public TipMotor motor;
  //atribut instanta  de tip TipCulori
```

```
  public TipCulori culoare;
}
```

La definirea enumerarii se are in vedere:

- ✓ simboluri sau constantele din enumerare sunt de obicei definite cu majuscule (cum ar fi DIESEL, BENZINA, …) daca se au in vedere conventiile de nume recomandate de Java;
- ✓ simbolurile din enumerari nu sunt valori int sau String;
- ✓ enumerarile delarate la nivel global pot fi definite DOAR default sau public (si NU private sau protected), dar in acest ultim caz, in propriile lor fisier .java (aceeasi regula vallabila si pentru clase);
- ✓ enumerarile declarate intr-o alta clasa pot avea modificatori de acces (private, public, default, protected) care controleaza vizibilitatea enumerarii in afara clasei parinte.

Pe baza descrierii anterioare, enumerarile sunt folosite pentru a controla valorile posibile ale unei variabile. Deci, daca vrem sa initializam atributul *motor* al instantei trebuie sa ne folosim numai constante din enumerarea TipMotor. Pentru enumerari definite in alte clase programatorul trebuie sa foloseasca numele complet, care include numele clasei parinte.

```
public class Main {
  public static void main(String[] args) {
    Vehicul v = new Vehicul();
    v.motor = TipMotor.DIESEL;

    //erori compilare:
    //v.motor = "DIESEL";  //eroare
    //v.motor = 1;         //eroare


        //referire completa:  nume_clasa.nume_enumerare.simbol
    v.culoare = Vehicul.TipCulori.GREEN;
    //compiler eroare
    //v.culoare = TipCulori.GREEN; //eroare
  }
}
```

O caracteristica interesanta a enumerarilor este ca simbolurile sale pot fi usor convertite la o valoare String egala cu numele simbolului. Codul secventa urmatoare:

```
    System.out.println("The vehicle motor type is "+v.motor);

    String motorType = TipMotor.ELECTRIC.toString();
    System.out.println("The motor type is "+motorType);
```

genereaza mesajul:

```
The vehicle motor type is DIESEL
The motor type is ELECTRIC
```

In Java enumerarile sunt clase:

In alte limbaje de programare cum ar fi C sau C++, enumerarile sunt colectii de simboluri care au asociat un ID unic numeric. Aceasta descriere nu se poate aplica si aici, pentru ca in Java enumerarile sunt clase. Simbolurile enumerarilor sunt atribute statice si constante ce reprezinta instante ale clasei din spatele enumerarii. Din aceasta perspectiva putem presupune ca simbolul DIESEL este definit de masina virtuala cu declaratie (aceasta este o presupunere care nu este foarte departe de adevar):

```
//expresia NU este valida in JAVA
//doar pentru a intelege simbolurile din enumerari
public static final TipMotor DIESEL = new TipMotor("DIESEL");
```

Deoarece enumerarile sunt clase, inseamna ca avem posibilitatea sa definim in interiorul acesto structuri (in afara de simbolurile sale):

- ✓ variabile de instanta
- ✓ metode de acces sau de prelucrare;
- ✓ constructori.

Daca vrem sa definim in interiorul unei enumerari mai mult decât simbolurile, intelegand prin asta alte valori sau rutine interne de prelucrare, putem atinge acest obiectiv datorita faptului ca structurile de tip enumerare sunt in Java clase.

Revenind la scenariul definit de clasa Vehicul, definim o serie de specificatii care sa justifice prelucrarea enumerarii din punct de vedere al unei clase:

- ✓ se defineste o variabila asociata fiecare tip de motor care va stoca un cod unic numeric;
- ✓ atributul cu rol de cod este protejat prin definirea acestuia privat;
- ✓ se defineste o metoda care da acces in mod citire la cod pentru a-I afisa valoarea;
- ✓ este nevoie de un constructor, deoarece fiecare simbol TipMotor are propriul cod unic.

Implementand cerintele anterioare, enumerarea TipMotor devine:

```
//enumerare definita independent
//ATENTIE ; din final este optionala
enum TipMotor{
  //fiecare simbol este creat apeland constructorul enumerarii
  DIESEL(10),BENZINA(20),HYBRID(30),ELECTRIC(40);

  //variabila de instanta pentru cod
  private int cod;

  //constructor
  TipMotor(int codValue){
     cod = codValue;
  }
```

```
  //metoda din enumerare
  public int getCode(){
     return cod;
   }
// terminare enumerare
}
```

In ciuda faptului ca enumerarea TipMotor arata ca o clasa, aceasta nu este una obisnuita, deoarece:

- ✓ modificatorii de acces public si protected **NU** sunt permisi pentru constructor;
- ✓ **NU** se pot crea instante prin apelul direct al constructorului (de exemplu new TipMotor(30));
- ✓ definirea simbolurilor enumerarii trebuie sa reprezinte prima declaratie din clasa;
- ✓ se poate apela connstructorul enumerarii DOAR numai când se definesc simbolurile (de exemplu, DIESEL(10)), chiar daca sintaxa este diferita pentru Java;
- ✓ este posibila supraincarcarea constructorilor din enumerare si definirea lor cu orice numar de parametri;

In acest caz, simboluri din enumerare se comporta ca obiecte si prin intermediul lor se pot apela metode din clasa. Variabilele de tip **TipMotor** reprezinta referinte la obiecte constante (simbolurile din enumerare). Secventa urmatoare:

```
System.out.println("Tipul motorului este " + v.motor +
      " iar codul acestuia este "+v.motor.getCode());
```

afiseaza:

```
Tipul motorului este DIESEL iar codul acestuia este 10
```

<u>Cum se parcurge lista de simboluri dintr-o enumerare:</u>

Fiecare enumerare are o metoda statica, **values()** , utilizata pentru a itera peste constantele/simbolurile sale. Secventa urmatoare:

```
System.out.println("Simbolurile din TipMotor sunt ");
for(TipMotor et : TipMotor.values())
{
   System.out.println(et + " cu codul "+et.getCode());
}
```

genereaza:

```
Simbolurile din TipMotor sunt
DIESEL cu codul 10
BENZINA cu codul 20
```

HYBRID cu codul 30
ELECTRIC cu codul 40

Ce reprezinta "constant specific class body":

Enumerarile sunt liste de obiecte constante care reprezinta un set limitat de valori. Aceste valori sunt semnificative intr-un context foarte specific.

In ciuda faptulcui ca enumerarile sunt clase, programatorii mentin complexitatea solutie la un nivel simplu, deoarece aceste structuri sunt un tip special de clasa cu un rol foarte specific. Cele mai multe dintre metodele dintr-o enumerare, daca exista, sunt simple (cum ar fi functiile accesor: get si set) si ofera solutii generice pentru simbolurile enumerarii.

Daca exista situatii care necesita mai mult decât accesul la valorile atributelor, se recomanda definirea unei noi clase si reanalizarea arhitecturii solutiei sau implementarea de solutii simple.

Pentru exemplul anterior, se adauga o noua specificatie:

- ✓ enumerarea ofera o metoda utilizata pentru a determina daca tipul motorului polueaza sau nu;

Daca analizam tipurile de motoare putem vedea ca toate sunt poluatori cu exceptia celui electric. Deci, solutia este de a defini o metoda care va testa tipul motorului:

```
enum TipMotor{
  //fiecare simbol este definit prin apelul constructorului
  DIESEL(10),BENZINA(20),HYBRID(30),ELECTRIC(40);

  ...

  public boolean isPoluant()
  {
    if(this.toString().equals("ELECTRIC"))
      return false;
    else
      return true;
  }
// final enumerare
}
```

Dupa cum se observa, codul devine putin prea complex pentru o enumerare. Pentru a pastra lucrurile simple, Java ofera o alta solutie numita **"constant specific class body"** (evit sa traduc acest termen, insa el poate fi interpreta ca o specializare a prelucrarilor pentru un simbol)**.** Aceasta reprezinta o situatie in care programatorii pot defini o implementare particulara (specializare) a unei metode pentru o anumita constanta/simbol din enumerare.

Astfel, exemplul anterior se modifica prin definirea unei forme supraincarcate a metodei isPoluant() asociata simbolului ELECTRIC.

```
//enumerated list declared as its own class
//REMEMBER the final ; is optional
enum TipMotor{
  //fiecare simbol este creat apeland constructorul enumerarii
  DIESEL(10),
  BENZINA(20),
  HYBRID(30),
  ELECTRIC(40){
    //constant specific class body
    //implementare particulara a unei metode
```

```
    //supradefineste implementarea generica
    public boolean isPoluant(){
       return false;
    }
};  // ; este OBLIGATORIE cand urmeaza cod

//variabila de instanta
private int cod;

//constructor
TipMotor(int codValue){
   cod = codValue;
}

//metoda
public int getCode(){
   return this.cod;
}

//implementarea generica a metodei
public boolean isPoluant(){
      return true;
}
}
```

In exemplul anterior, implementarea particulara a metodei isPoluant() pentru simbolul ELECTRIC are prioritate fata de implementarea generica, furnizand o valoare specifica acestui simbol.

Memoria Stack-ul(Stiva) si memoria Heap

Pentru a avea o intelegere profunda a progamarii orientate obiect in Java sau in orice alt limbaj orientat obiect (cum ar fi C #),trebuie sa stii cum sunt gestionate lucurile intern de catre procesul Java si de JVM (Java Virtual Machine). Desigur, sintaxa si implementare principiilor POO (Programare Orientata Obiect) in Java sunt importante, dar vei avea o imagine mult mai clara cu privire la resursele necesare, memorie, performanta, transferul parametrilor, fire de executie si de eliberare a memoriei sau colectare a gunoiului (garbage collection), daca ai iti pui intrebari dincolo de *Cum fac asta ?* sau *Cum scriu asta ?*. Intrebari reale trebui sa fie *Cum* sau *De ce se intampla asa ?* (desigur, de la un punct, trebuie sa iei lucrurile asa cum sunt si sa mergi mai departe).
In acest tutorial voi descrie modul in care variabilele aplicatiei sunt gestionate, in ceea ce priveste locul unde sunt depozitate (Stack – Stiva sau Heap) si pentru cat timp.

Ce este Stack-ul (Stiva) si Heap-ul:

Pentru a mentine lucrurile la un nivel simple (daca ai cunostinte de programare in limbaj de asamblare, vei vedea ca prezentarea urmatoare reprezinta o abordare superficiala;
Pornind de la ipoteza ca aplicatia prelucreaza date ce sunt stocate in anumite zone din RAM (Random Access

Memory). Aceste zone se numesc:

Stack:

- ✓ un spatiu de memorie rezervat pentru proces (aplicatie) de catre sistemul de operare;
  dimensiunea stivei este fixa si se stabileste, in faza de compilare, pe baza declaratiilor de variabile si alte optiuni de compilare;
- ✓ este important sa se inteleaga faptul ca stiva este limitata, iar dimensiunea ei este fixa (un proces inceput, nu poates chimba dimensiunea stivei sale);
- ✓ de cele mai multe ori, stiva este utilizat pentru a stoca variabilelor functiilor (argumente de intrare si variabile locale);
- ✓ fiecare metoda are propia stiva (o zona in stiva procesului), inclusiv metoda speciala **main**, care este de asemenea o functie;
- ✓ stiva unei metode exista doar pe durata de viata a acestei metode: din momentul apelarii pana in momentul terminarii functiei (return sau o exceptie); acest comportament se datoreaza chiar faptului ca stiva este limitata si spatiul este pus la dispozitia urmatoarei metode.

Heap:

- ✓ un spatiu de memorie gestionate de sistemul de operare si utilizate de catre procese pentru a obtine spatiu suplimentar la executie (run-time);
- ✓ acest spatiu exista la un nivel global, ceea ce inseamna ca orice proces poate folosi (desigur, procesele nu pot citi sau scrie intr-o zona din Heap rezervata altui proces);
- ✓ rolul acestui memorii este de a oferi resurse suplimentare de memorie proceselor care au nevoie de spatiu suplimentar la run-time (de exemplu, va puteti gandi la o simpla aplicatie Java, care construieste un vector a carui dimensiune si elemente sunt primite de la consola);
- ✓ spatiul necesar in Heap este determinat de functia *new* (**este aceeasi functie folosita pentru a crea obiecte in Java**).

Pachete(Package) de Clase

Pachetele sunt un spatiu de nume care organizeaza un set de clase si interfete. Conceptual putem gindi ca un pachet este similar erarhiei a mapelor din calculator.

Notiuni:

- ✓ Este o colectie de clase; grupeaa clasele folosite impreuna.
- ✓ Folosirea pachetelor face mai usor lucrul cu multe clase.
- ✓ Fiecare pachet are propriul lui spatiu de nume.
- ✓ Utilizarea pachetelor in Java se face prin doua cuvinte cheie: package si import.

Exemplu:

Folosirea clasei Administrator din alt pachet:
Import edu.zoo.personal.Administrator;
Sau pentru a folosi toate clasele din pachet:
Import edu.zoo.personal.*;

Pachete de clase:

- ✓ Bibliotecile de clase uzuale sunt organizate in pachete.
- ✓ Un pachet(Package) contine clase uzuale specifice unui anumit gen de aplicatii.

Exemple:

- *Java.lang* - pachetul de clase de baza ale java.
- *Java.io* – clase pentru transferuri datelor din/catre exterior.
- *Java.net* – clase de obiecte specifice conexiunii prin retea.
- *Java.swing* – clase de obiecte grafice pentru GUI(graphical user interface).
- *Java.swing.event* – clase de evenimente specifice GUI.

Utilizatorul isi poate construi propriile pachete de clase.