

5. MOSTENIRE. CLASE ABSTRACTE. INTERFETE. PACHETE

5.1. Necesitatea mostenirii si aspectele sale de baza.....	<u>2</u>
5.1.1. Inconveniente ale crearii unui nou tip de date pe baza uneia existente.....	<u>2</u>
5.1.2. Solutia eleganta: mostenirea.....	<u>2</u>
5.1.3. Relatia introdusa intre clase: terminologie si modalitati de reprezentare.....	<u>3</u>
5.1.4. Beneficiile mecanismului de mostenire.....	<u>3</u>
5.1.5. Cum apar ierarhiile de clase.....	<u>4</u>
5.1.6. Verificarea relatiei "is a".....	<u>5</u>
5.2. Detaliile tehnice ale mecanismului de mostenire.....	<u>6</u>
5.2.1. Libertati si restrictii.....	<u>6</u>
5.2.2. Cum se instantiaza o clasa derivata. Relatia intre constructori.....	<u>6</u>
5.2.3. Controlul accesului la membrii mosteniti.....	<u>7</u>
5.2.4. Redefinirea membrilor in subclase.....	<u>8</u>
5.2.4.1. Concepte.....	<u>8</u>
5.2.4.2. Redefinirea campurilor.....	<u>9</u>
5.2.4.3. Overriding: rescrierea implementarii metodelor mostenite.....	<u>9</u>
5.2.4.3.1. Concepte si particularitati.....	<u>9</u>
5.2.4.3.2. Refolosirea codului din implementarea clasei parinte.....	<u>11</u>
5.2.4.3.3. Clasa Object: overriding-ul este chiar recomandabil.....	<u>11</u>
5.2.4.3.4. Relatia overloading-overriding (supraincarcare-rescriere).....	<u>12</u>
5.2.5. Upcasting, downcasting si polimorfism.....	<u>12</u>
5.2.6. Mostenirea si membrii statici.....	<u>14</u>
5.2.7. Interzicerea mostenirii.....	<u>14</u>
5.3. Clase si metode abstracte: instantierea interzisa, implementarea obligatorie in subclase.....	<u>15</u>
5.4. Interfete: iata forma, implementati continutul.....	<u>15</u>
5.5. Pachete java (packages).....	<u>18</u>
5.5.1. Principii.....	<u>18</u>
5.5.2. Declararea apartenentei la un pachet.....	<u>18</u>
5.5.3. Numele pachetelor.....	<u>19</u>
5.5.4. Accesarea claselor ce fac parte dintr-un pachet.....	<u>19</u>
5.5.4.1. Implicatii ale apartenentei la pachet.....	<u>19</u>
5.5.4.2. Utilizarea directivei import.....	<u>20</u>
5.5.4.2.1. Principii. Forme ale directivei.....	<u>20</u>
5.5.4.2.2. Importarea unei clase punctuale.....	<u>20</u>
5.5.4.2.3. Importarea tuturor claselor dintr-un pachet.....	<u>21</u>
5.5.4.2.4. Importarea membrilor statici ai unei clase.....	<u>21</u>
5.5.5. Structura de pachete JRE.....	<u>22</u>
5.5.6. Mecanismul de incarcare a claselor.....	<u>22</u>
5.6. BIBLIOGRAFIE.....	<u>23</u>

5.1. Necesitatea mostenirii si aspectele sale de baza

5.1.1. Inconveniente ale crearii unui nou tip de date pe baza unuia existent

Sa presupunem ca am creat intr-o aplicatie o clasa numita Persoana avand definitia urmatoare:

```
class Persoana {
    String nume;

    public int getNume() { return nume; }
    public void setNume(String n){ nume = n; }
    public void prezентare() { System.out.println("Salut, eu sunt "+nume); }
}
```

Daca avem nevoie de o clasa Instructor care are toate caracteristicile unei persoane dar si cateva suplimentare, proprii, solutia nu este duplicarea de cod, ci reutilizarea intr-un fel sau altul a clasei Persoana. O posibila solutie este cea de mai jos, in care fiecare obiect Instructor incapsuleaza un obiect Persoana si deleaga catre ele metodele pe care acesta le suporta:

```
class Instructor{
    private Persoana p;
    public Instructor(String nume) { p=new Persoana(nume); }
    public int getNume{ return p.getNume(); }
    public String setNume(String n) { p.setNume(n); }
    public void predare(){ System.out.println("Instructorul "+p.getNume()+" sustine o lectie"); }
}
```

Ce observam aici este o incapsulare a unui obiect de tip Persoana in cel de tip Instructor si redirectionarea metodelor obiectelor de tip Instructor catre cele ale obiectului de tip Persoana incapsulat. In acest fel, instructorul "stie" sa faca tot ceea ce facea persoana (refolosind codul de acolo), utilizatorul unui obiect de tip Instructor nefiind conscient de acest artificiu.

Solutia, desi pare relativ rezonabila, sufera de doua dezavantaje majore:

- esfertul delegarii. Daca clasa Persoana contine 10 metode, programatorul ar fi fost nevoie sa efectueaza delegarea pentru fiecare dintre ele. Ar fi de dorit ca aceasta operatie sa se efectueze automat, ea nepresupunand creativitate ci simpla rutina
- tipurile de date Persoana si Instructor sunt percepute ca distincte de catre compilator si masina virtuala, deoarece nu putem profita cum se cuvine de pe urma functionalitatii comune a celor doua clase. Ce se intampla insa daca aplicatia noastra foloseste si o clasa Student (care la randul sau are nevoie sa refoloseasca codul din Persoana) si isi propune sa creeze un tablou populat cu obiecte de tip Persoana, Student sau Instructor? Ce tip de date va trebui sa aiba tabloul, astfel incat a) sa-l putem crea si sa accepte atat obiecte Instructor cat si Student, si b) sa putem apela oricarui dintre obiectele componente metodele pe care stim sigur ca le are? (getNum(), setNume())Sa nu uitam ca un tablou contine doar elemente de acelasi tip, sau care sunt compatibile cu tipul de date al elementelor tabloului (ex: byte poate fi folosit intr-un tablou de int).

Mecanismul de mostenie rezolva aceste doua probleme, dupa cum se va vedea in cele ce urmeaza.

5.1.2. Solutia eleganta: mostenie

Mecanismul de mostenie presupune "marcarea" clasei Instructor ca fiind derivata din Persoana, folosind cuvantul cheie **extends** ca mai jos:

```
class Instructor extends Persoana{
    public void predare(){ System.out.println("Instructorul "+nume+" sustine o lectie"); }
```

```

public static void main(String[] args){
    Instructor x = new Instructor();
    x.setNume("Mihai");           // metoda mostenita din clasa Persoana
    x.prezentare();              // metoda mostenita din clasa Persoana
    x.preda();                   // metoda proprie
}

```

Observam in codul de mai sus cum clasa *Instructor* preia automat membrii clasei *Persoana* – fapt evidentiat in doua moduri:

- ii putem apela obiectului de tip *Instructor* atat metodele proprii (*preda*) cat si cele mostenite din clasa *Persoana*, fara a mai fi nevoie sa efectuam delegarea acelor metode
- in metodele clasei *Instructor* putem folosi campul *nume* ca si cum ar fi fost declarat in aceeasi clasa; de fapt, el este la randul sau mostenit din clasa *Persoana*

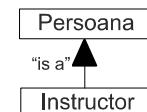
Constatam asadar o mult mai eleganta refolosire de cod pe care o ofera acest nou mecanism de mostenire. Acesta nu este insa singurul sau avantaj, dupa cum se va vedea in continuare.

5.1.3. Relatia introdusa intre clase: terminologie si modalitati de reprezentare

Cele doua clase ce participa intr-o relatia de mostenire sunt desemnate folosind urmatorii termeni:

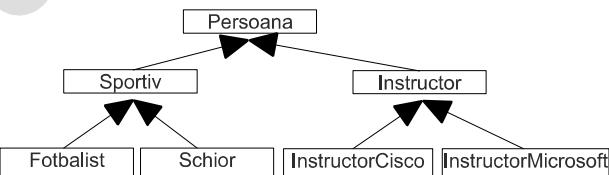
- **clasa parinte** sau **clasa de baza** - reprezinta clasa de origine, pe baza careia a fost creata cea de-a doua
- **subclasa** sau **clasa derivata** - reprezinta noua clasa creata pe baza celei parinte

Vizual, relatia dintre ele se reprezinta ca in figura alaturata: clasa parinte situata deasupra, iar sageata orientata de la subclasa catre clasa parinte.



Relatia dintre cele doua clase este in general descrisa cu sintagma “is a” (este un/o) – in sensul in care un instructor *este o Persoana* (adica se comporta ca ea in toate privintele). Implicatiile in programare vor fi detaliate ulterior.

O clasa parinte poate avea mai multe subclase; acestea pot fi la randul lor extinse, subclasele lor pot avea si ele subclase etc, astfel incat rezulta in final o intreaga **ierarhie de clase**. Oricare doua clase aflate pe aceeasi verticala se gasesc in relatia “is a”, aceasta fiind tranzitiva. In figura alaturata, un *Fotbalist* “is a” *Sportiv* si de asemenea “is a” *Persoana*, iar un *InstructorCisco* la randul sau se afla in relatia “is a” atat cu *Instructor* cat si cu *Persoana*.



Nota: clasele aflate pe ramuri diferite nu se afla in nicio relatie directa! Nu putem afirma ca un fotbalist “is a” InstructorCisco.

In cadrul aceliasi verticala a ierarhiei de clase, pe masura ce coboram gasim clase din ce in ce mai specializate: un Sportiv este un anumit tip particular de Persoana, precum un Fotbalist este un Sportiv specializat pe un anumit domeniu. Chiar daca un Fotbalist poate face acelasi lucru ca o Persoana, e posibil sa se faca altfel, si in plus are abilitati suplimentare fata de o Persoana obisnuita. Pe aceeasi logica, pe masura ce urcam in ierarhia de clase vom intalni clase din ce in ce mai generale, mai abstracte.

5.1.4. Beneficiile mecanismului de mostenire

Mecanismul de mostenire nu trebuie privit exclusiv ca unul de refolosire de cod. Desi acesta este unul dintre beneficiile pe care le ofera, exista si un alt doilea, superior ca importanta: crearea de tipuri de date inrudite.

Din momentul in care am marcat clasa *Instructor* ca fiind derivata din *Persoana*, cele doua clase vor fi percepute de catre compilator si masina virtuala ca fiind inrudite. Relatia dintre ele (“is a”) nu numai ca sta in picioare la nivel de exprimare umana, ci este de acum incolo valabila si la nivel de tipuri de date: un obiect *Instructor* are toate capabilitatile unuia de tip *Persoana* si deci poate fi folosit fara riscuri in locul acestuia. Prezentam urmatorul exemplu in acest sens:

```

class Mostenire{
    static void intalnire(Persoana p){
        p.prezentare();
    }
    public static void main(String[] args){
        Persoana p = new Persoana();
        Instructor i = new Instructor();

        // 1. O metoda care primeste ca argument o Persoana accepta la apelare o valoare
        //     de tip Instructor
        intalnire(p);
        intalnire(i);
        // 2. Un tablou de Persoana poate contin si obiecte de tip Instructor
        Persoana[] grup = new Persoana[2];
        grup[0] = p;
        grup[1] = i;
        // 3. un obiect de tip Instructor poate fi controlat cu o referinta de tip Persoana
        Persoana p = new Instructor();
    }
}

```

Nota: in exemplul de mai sus, scenariile 1 si 2 sunt de fapt cazuri particulare ale lui 3: in ambele vom ajunge sa referim un obiect Instructor folosindu-ne de o referinta de tip Persoana.

Constatam ca, privind dintr-un anumit unghi, mostenirea creeaza familii de tipuri de date custom care au o relatie de inclusiune: instructiunea `Persoana = new Instructor()` functioneaza gratie relatiei "is a" care face ca valoarea din dreapta sa "incapa" in (sa fie compatibila cu) variabila din stanga, asemănător instrucțiunii `long L = 14` de la tipurile de date primitive. Clasa Persoana devine astfel un tip de date "umbrela", in locul sau putand fi folosite oricare dintre subtipuri.

Dupa cum observam, intrudirea claselor are o implicatie imediata: validarile pe care le poate face acum compilatorul. Prin faptul ca un instructor este o persoana, obiectele de tip `Instructor` prezinta garantia pentru compilator si masina virtuala ca pot fi folosite in locul celor de tip `Persoana`.

5.1.5. Cum apar ierarhiile de clase

Din felul in care au fost prezentate lucrurile pana acum s-ar putea deduce ca ierarhiile de clase apar derivand (extinzand) clase deja scrise, ceea ce nu este intotdeauna adevarat. Exista cel putin doua scenarii ce determina aparitia de ierarhii:

- avem o clasa deja scrisa si constatam ca avem nevoie de una cu capacitatii asemanatoare, dar mai specializata (cu eventuale campuri/metode suplimentare fata de cea parinte, si cu eventuale manifestari diferite ale metodelor mostenite). Acest scenariu este cel prezentat pana acum, care actioneaza "de sus in jos"
- avem doua sau mai multe clase care constatam ca au *unele* capacitatii asemanatoare. Pentru a putea profita de aceste elemente comune, le extragem sub forma unei clase parinte care va fi derivata mai apoi de celelalte clase. Acest scenariu actioneaza "de jos in sus" si este detaliat in continuare.

Sa luam exemplul animalelor de companie: atat pisica cat si cainele merg, mananca si se joaca, in schimb pisica muauna iar cainele latra. Am putea defini doua clase diferite:

<pre> class Pisica { String nume; public void merge() {} public void mananca() {} public void miauna() {} } </pre>	<pre> class Caine { String nume; public void merge() {} public void mananca() {} public void latra() {} } </pre>
---	---

Observam insa ca ambele tipuri de date au membri comuni: campul `nume` si metodele `merge()` si `mananca()`. De ce sa existe doua tipuri de date complet diferite (cu duplicarea de cod aferenta) atata timp cat ele au o parte comună considerabila? Putem defini un tip de date `Animal` care contine acele caracteristici comune, urmand sa cream tipurile de date `Caine` si `Pisica` prin particularizarea tipului `Animal`; procedeul este scalabil, pentru ca putem crea in continuare si alte clase care extind clasa `Animal` si care mostenesc de la aceasta capacitatatile comune tuturor animalelor:

```
class Animal {
    String nume;
    public void merge() { System.out.println(nume+" se plimba nepasator"); }
    public void mananca() { System.out.println(nume+" mananca cu pofta"); }
}
```

Odata creat tipul *Animal*, *Caine* va extinde acest tip de date si va adauga propria metoda *latra()*, iar *Pisica* va adauga metoda *miauna()*:

```
class Caine extends Animal {
    public void latra() { System.out.println(nume + " zise: Ham!"); }
}
class Pisica extends Animal {
    public void miauna() { System.out.println(nume + " zise: Miau!"); }
}
```

Aceasta abordare are urmatoarele avantaje:

- am eliminat codul duplicat din cele doua clase prin integrarea sa in clasa parinte *Animal*, ceea ce duce la propagarea lui automata in subclase
- putem exploata partea comună a celor două tipuri de date folosind tipul “umbrelă”:

```
public static void main(String[] a){
    Pisica p = new Pisica();
    Caine c = new Caine();
    Animal[] pets = {p,c};

    pets[0].merge();
    pets[1].mananca();
}
```

5.1.6. Verificarea relatiei “is a”

Java ofera operatorul **instanceof** dedicat verificarii relatiei “is a” intre un obiect si o clasa. Sintaxa sa este urmatoarea:

```
referinta instanceof Clasa
```

Operatorul produce *boolean* indicand daca clasa se regaseste in ierarhia de inheritance a obiectului, la orice nivel! Spre exemplu:

```
class Om{}
class Sportiv extends Om{}
class Schior extends Sportiv{}
.....
Schior s = new Schior();
// urmatoarele trei expresii produc true:
s instanceof Schior
s instanceof Sportiv
s instanceof Om
```

Nota: *instanceof* produce true si pentru relatiile “is a” create cu ajutorul interfetelor - vezi sectiunea dedicata acestora

5.2. Detaliile tehnice ale mecanismului de mostenire

5.2.1. Libertati si restrictii

Cea mai importanta restrictie a mecanismului de mostenire in Java este ca **nu exista mostenire multipla** (de exemplu, o clasa Instructor care ar mosteni simultan caracteristicile clasei Persoana si ale clasei Angajat). In definitia unei subclase, dupa cuvantul cheie *extends* poate urma o singura alta clasa; o clasa care deja are un parinte nu mai poate extinde alta.

Nota: un efect echivalent cu al mostenirii multiple se obtine in Java folosind un tip special de clasa, numit **interfata**, tratat ulterior in acest material.

Mecanismul de mostenire ii permite programatorului un numar de libertati care sa acopere necesitatile numeroaselor scenarii ce apar in viata reala:

- **controlul accesului la membrii mosteniti.** O clasa parinte poate avea campuri sau metode care tin de "bucataria" sa interne si care nu dorim sa fie accesibili subclase; exista mecanisme de control in acest scop
- **adaugare de noi membri in subclase.** Subclasele sunt versiuni mai specializate ale claselor parinte, iar aceasta poate inseamna ca contin informatie suplimentara sau au metode in plus. Spre exemplu, clasa Instructor poate avea atat metodele vorbeste() si mananca() mostenite din Persoana, cat si metodele proprii pred() sau scrieMaterialeDeCurs()
- **rescrierea metodelor mostenite.** O subclasa fiind o varianta specializata a clasei parinte, este posibil sa doreasca sa ofere o versiune specializata a unei metode pe care o mosteneste de la parinte. Spre exemplu, un OmDeCultura va mosteni metoda vorbeste() din Persoana, insa implementarea sa vom dori probabil sa fie diferita
- **manifestarea individualitatii subclaselor** unei clase parinte care au fiecare cate o versiune a unei metode rescrise. Spre exemplu, daca clasele *Chinez*, *Englez* si *Roman* deriva din *Persoana*, vor avea fiecare propria implementare a metodei *vorbeste()*; daca cream un tablou de obiecte *Persoana*, ni s-ar parea normal ca, atunci cand il parcurgem, chiar daca referim fiecare obiect cu o referinta de tip *Persoana*, el sa "vorbeasca pe limba lui"
- **interzicerea mostenirii.** Uneori dorim ca o anumita metoda sa nu poate fi rescrisa in subclase, sau poate chiar ca o anumita clasa sa nu poata fi extinsa

Toate aceste grade de libertate vor fi detaliate in continuare.

5.2.2. Cum se instantiaza o clasa derivata. Relatia intre constructori

Mecanismul de mostenire este - cel putin in parte - o modalitate de a automatiza in mod flexibil solutia delegarii prezentata in prima sectiune a materialului de fata. Atunci cand se instantiaza o subclasa, noul obiect preia caracteristici din clasa parinte - ceea ce inseamna ca, in fundal, masina virtuala acceseara si foloseste definitia acelei clase. Preluarea nu se face selectiv, ci integral: fiecare obiect al subclasei include de fapt un obiect "miez" de tip parinte, cu tot ceea ce contine acesta.

La crearea unei instance de subclasa este necesara initializarea tuturor campurilor - atat cele mostenite de la clasa parinte, cat si cele suplimentare ale subclasei. Pentru prima categorie exista deja constructor(i) in clasa parinte si deci ne dorim sa putem refolosi acel cod. Reamintim insa cititorului ca constructorul se manifesta doar partial ca o metoda a clasei – in particular, el **nu este mostenit**. De aceea, constructorul parinte trebuie apelat explicit din cadrul constructorului subclasei.

Observatie: la instantierea unei subclase ruleaza asadar ambii constructori - cel parinte si cel al subclasei, in aceasta ordine.

Apelul catre constructorul parinte este guvernat de urmatoarele reguli:

- se realizeaza folosind cuvantul cheie **super** urmat de eventualele argumente pasate constructorului parinte
- **trebuie sa fie prima instructiune din constructorul subclasei**
- daca programatorul nu introduce explicit acest apel, compilatorul va incerca sa introduca automat apelul catre constructorul parinte fara argumente. Absenta unui astfel de constructor in clasa parinte va duce la o eroare de compilare

Nota: concluzionam ca, daca clasa parinte are exclusiv constructori cu argument, programatorul este obligat sa scrie explicit in constructorul subclasei apelul catre constructorul parinte; compilatorul nu mai poate ajuta in acest caz.

Folosind acest mecanism, initializarea campurilor obiectului subclasei poate fi realizata disociat: campurile care tin de clasa parinte vor fi initializate de constructorul parinte (refolosind astfel cod), iar campurile proprii subclasei vor fi initializate de catre constructorul acesteia.

```

class Animal {
    String nume;
    public Animal(String n) { nume = n; }
}
// initializare disociata, cu refolosire de cod
class Molie extends Animal{
    boolean imunitateLaNaftalina;
    public Molie(String n, boolean i){ // constructorul primeste valori pentru ambele campuri!
        super(n);                      // initializarea campurilor mostenite din clasa Animal
        imunitateLaNaftalina = i;       // initializarea campurilor proprii
    }
}
// cand constructorul parinte are argumente nu putem omite apelul catre el in subclasa!
// de aceea, urmatoarele clase nu compileaza:
class Caine extends Animal{}

class Cartita extends Animal {
    int varsta;
    public Cartita() {
        nume = "Daredevil";
        varsta = 3;
    }
}

```

Explicatie: din cauza definirii unui constructor cu parametri in clasa *Animal*, compilatorul nu va mai introduce automat constructorul default in aceasta clasa (cel fara argumente). Ca urmare:

- daca programatorul nu scrie constructor in subclasa, compilatorul va incerca sa-l introduca automat pe cel default (fara argumente), iar in cadrul lui apelul catre constructorul parinte, de asemenea fara argumente. Neexistand un astfel de constructor in clasa *Animal*, codul nu va compila
- daca programatorul scrie constructorul subclasei ca in clasa *Cartita* de mai sus, incercand initializarea “de mana” a tuturor campurilor, compilatorul va incerca sa insereze ca prima instructiune din acest constructor apelul catre cel parinte fara argumente, de unde rezulta iarasi o eroare

Nota: nu exista nicio relatie obligatorie intre numarul si tipurile de date ale argumentelor din constructorul parinte si din cel al subclasei. In general insa, constructorul de subclasa are argumente in plus, deoarece are de initializat campurile suplimentare introduse in subclasa.

Observatie: apelarea constructorului parinte din cel al subclasei este o necesitate; dupa cum se va vedea, atunci cand un camp are nivel de acces private in clasa parinte, el nu este accesibil in subclase si atunci constructorul parinte ramane singura sa posibilitate de initializare!

5.2.3. Controlul accesului la membrii mosteniti

S-a afirmat anterior ca un obiect al subclasei contine integral un obiect “miez” de tip parinte. Deseori dorim insa ca o subclasa sa nu aiba acces la toti membrii clasei parinte - poate ca unele campuri sau metode tin de bucataria interna a acelei clase si nu trebuie folosite direct in subclase. Aceasta presupune sa putem interzice accesul codului din subclasa la o parte din membrii prezenti in “miez”, iar acest lucru se realizeaza in baza nivelului de acces al membrilor clasei parinte, stabilit prin intermediul modificatorilor de acces deja cunoscuti:

Modificator de acces	Nivel de acces	Membrul este vizibil din afara obiectului?	Membrul este vizibil in subclase?
public	public	DA	DA
private	private	NU	NU

protected	protected	DA, pentru clase din acelasi pachet; NU in rest	DA
(lipseste)	default (package-private)	DA, pentru clase din acelasi pachet; NU in rest	DA, pentru clase din acelasi pachet; NU in rest

Iata un exemplu care ilustreaza toate cazurile posibile. Pentru simplitate au fost folosite numai campuri, insa concluziile se aplică în mod accesului la metode:

clasa A	clasa B aflata in acelasi pachet cu A	clasa C aflata in alt pachet decat A si B
<pre>class A{ public int a; private int b; protected int c; int d; }</pre>	<pre>class B extends A{ void test(){ a++; // ok b++; // nu merge c++; // ok d++; // ok } }</pre>	<pre>class C extends A{ void test(){ a++; // ok b++; // nu merge c++; // ok d++; // nu merge } }</pre>

Aveți tendința de a spune că “un membru privat nu se mosteneste”, deoarece el nu “se vede” în subclasa; tehnic, el este prezent în obiectul subclasei, însă nivelul sau de acces nu permite accesarea sa directă în cadrul subclasei. Aceasta nu înseamnă că el nu poate fi utilizat - indirect - în subclasa! Priviți exemplul următor:

```
class Persoana {
    private String nume;
    public String getNume(){ return nume; }
    public void setNume(String n){ nume = n; }
}
class Student extends Persoana{
    public void prezентare(){
        System.out.println("Salut, ma cheama "+getNume());
        // System.out.println("Salut, ma cheama "+nume()); → aceasta linie nu ar fi compilată,
        // deoarece campul nume nu e direct accesibil în subclasa
    }
    public static void main(String[] args){
        Student s = new Student();
        s.setNume("Mihai"); // metoda setNume() e publică în clasa parinte și deci mostenită;
        // ea memorează informația în campul nume prezent în “miezul” obiectului Student
        s.prezентare();      // numele setat a fost memorat în obiect și este acum folosit
    }
}
```

În clasa *Persoana*, campul *nume* este declarat *private*, asadar inaccesibil în clasa *Student*. Pe de altă parte, getterul și setterul său sunt publice, ceea ce determină accesibilitatea lor în clasa *Student*, ele continuând să acioneze asupra campului *nume* din “miezul” de tip *Persoana*!

Nota: dacă nu existau getteri și setteri, singura posibilitate de a initializa campul *nume* ramanea apelul catre constructorul parinte din cadrul constructorului clasei *Persoana*.

5.2.4. Redeclararea membrilor în subclase

5.2.4.1. Concepte

Adaugarea de noi membri în subclase cu greu poate fi privita ca pe un grad de libertate, deoarece tine de insasi esenta mecanismului de mostenire. Conform acestuia, subtipurile de date sunt mai specializate decat cele parinte, iar aceasta specializare inseamna deseoři tocmai adaugiri. Modul de punere în practica al acestui mecanism de imbogatire al unui tip de date a fost deja exemplificat în secțiunile situații anterioare.

Vom aborda însă un alt caz - cel al suprapunerilor:

- cand o metoda din clasa parinte este redeclarata in subclasa, avem de-a face cu un overriding (rescriere) a implementarii acelei metode. Varianta rescrisa face subiectul unui set important de restrictii - vezi capitolul dedicat overriding-ului
- cand un camp din clasa parinte este redeclarat in subclasa, el mascheaza campul din clasa parinte, acesta din urma existand insa in continuare!

5.2.4.2. Redefinirea campurilor

A redifini un camp inseamna a declara in subclasa un camp cu acelasi nume ca cel din clasa parinte. Java permite redifinirea indiferent de nivelul de acces al campului in clasa parinte; in plus, noul camp poate avea tip de date si modificator de acces diferite fata de cel original!

Efectul redifinirii este ca noul camp il va masca pe cel vechi (atunci cand este folosit cu numele sau scurt); campul original exista insa in continuare in "miezul" parinte si poate fi accesat in doua moduri:

- indirect, folosind eventuali getteri/setteri mosteniti din clasa parinte
- direct, folosind cuvantul cheie super in cadrul metodelor subclasei, daca nivelul de acces o permite

```
public class Parinte {
    protected int x;
    public void setX(int x) { this.x = x; }
    public int getX() { return x; }
}
class Subclasa extends Parinte{
    public boolean x; // redifinire cu alt tip de date si alt modificator de acces
    public void f(){
        System.out.println("x-ul din subclasa: "+x);
        System.out.println("x-ul din parinte: "+super.x); // nu ar fi mers daca x era
                                                       // private in clasa parinte
    }
    public static void main(String[] args) {
        Subclasa s = new Subclasa();
        s.x = true; // x-ul din subclasa
        s.setX(4); // x-ul din miez
        System.out.println(s.x); // true (se afiseaza x-ul din subclasa)
        System.out.println(s.getX()); // 4 (se afiseaza x-ul din miez)
        s.f(); // x-ul din subclasa: true
               // x-ul din parinte: 4
    }
}
```

5.2.4.3. Overriding: rescriverea implementarii metodelor mostenite

5.2.4.3.1. Concepte si particularitati

Este posibil sa modificam implementarea unei metode mostenite in subclasa astfel incat sa se potrivesca noilor necesitati/cerinte din subclasa. Procedeu se numeste **overriding** (il vom numi *rescrivere*) si nu trebuie confundat cu overloading-ul (supraincarcarea), fiind guvernat de legi diferite.

Nota: riguros vorbind, termenul de overriding nu se traduce ca rescrivere, insa termenul romanesc ales este potrivit pentru a descrie ce se intampla cu implementarea metodei.

Motivele pentru care o subclasa ar dori sa rescrie o metoda primita din parinte sunt diverse:

- implementarea metodei difera de la o subclasa la alta. Spre exemplu, data fiind clasa parinte *Medicament*, implementarea metodei *administrare()* va difera pentru subclasele *Aspirina*, *Algocalmin* etc
- implementarea metodei din subclasa trebuie sa foloseasca membri care au fost adaugati chiar in subclasa, si deci inexistenti in implementarea parinte (vezi exemplul urmator)

A face override unei metode inseamna sa redifini metoda in subclasa oferindu-i o noua implementare, ca in exemplul urmator:

```

class Animal {
    String nume;
    public void scoateSunet() {
        System.out.println("Animalul "+nume+" scoate sunete neinteligibile");
    }
}
class Tantar extends Animal {
    float frecventaDeLucru;
    public void scoateSunet() {
        System.out.println("Tantarul " + nume + " bazaie pe frecventa " + frecventaDeLucru);
    }
}

```

Varianta rescrisa a metodei (cea din subclasa) trebuie sa se supuna unui set strict de reguli, ea trebuind sa aiba:

- acelasi nume
- aceeasi succesiune a tipurilor de date ale parametrilor
- acelasi tip de date de intoarcere
- acces cel putin la fel de permisiv ca metoda originala
- o lista de exceptii declarate care sa nu contine elemente in plus fata de metoda parinte (vezi capitolul despre exceptii in a doua parte a cursului)

Nota: ne punem problema rescrierii numai pentru metode care au in clasa parinte un nivel de acces mai permisiv decat private, ca sa fie astfel accesibile in subclasa. Daca metoda parinte este private, atunci in subclasa avem libertate totala in alegerea parametrilor enumerati mai sus, deoarece are loc o declarare de metoda noua, nu o rescriere.

```

class Animal{}
class Mamifer extends Animal{}
class Pisica extends Mamifer{}

class Parinte{
    protected int f(int x, Mamifer m){...}
    protected Mamifer g(){...}
}

class Subclasa extends Parinte{
    // variante de overriding invalide
    private int f(int x, Mamifer m){...}           // metoda devine inaccesibila in exterior
    protected boolean f(int x, Mamifer m){...}       // varianta rescrisa produce alt tip de date
    protected byte f(int x, Mamifer m){...}          // idem - tip de date de intoarcere incompatibil
    protected int f(byte x, Mamifer m){...}          // rezulta overloading! (vor exista ambele metode)
    protected int f(int x, Pisica p){...}            // idem - overloading
    protected int f(long x, Mamifer m){...}          // idem - overloading
    protected int f(int x, Animal m){...}             // idem - overloading

    // variante de overriding corecte
    protected int f(int x, Mamifer m){...}           // nicio schimbare in semnatura
    public int f(int x, Mamifer a){...}               // nivel de acces mai permisiv
}

```

Nota: pentru simplificare, in exemplul de mai sus au fost omise implementarile metodelor; in mod normal, definitia unei metode care produce int nu va compila daca corpul sau nu contine o instructiune return compatibila.

Observam ca majoritatea regulilor ce guverneaza overriding-ul pot fi deduse prin urmatoare logica: avand in vedere ca un obiect *Subclasa* “is a” *Parinte*, programatorul are dreptul de a scrie *Parinte p = new Subclasa()*. Implementarea din subclasa nu trebuie sa produca surprize celui care manevreaza un obiect al subclasei prin intermediul unei referinte parinte, prin abaterea de la semnatura metodei parinte (ex: furnizand un alt tip de date de iesire).

5.2.4.3.2. Refolosirea codului din implementarea clasei parinte

Aşa cum, în cazul redeclarării de campuri în subclase, putem accesa din metodele subclasei atât campul subclasei, cât și cel parinte, există un procedeu asemănător aplicat metodelor. În cazul lor aceasta posibilitate este și mai importantă, deoarece apelarea implementării din metoda parinte în subclasa poate însemna o prețioasă refolosire de cod.

În măsură în care nivelul de acces din clasa parinte o permite, implementarea parinte este accesibilă în subclasa folosind cuvântul cheie **super** ca prefix pentru numele metodei:

```
class Eveniment {
    public void pregatire(){ System.out.println("Se anunta evenimentul in media"); }
}
class Oscaruri extends Eveniment{
    public void pregatire(){
        super.pregatire(); // refolosim codul din clasa parinte
        System.out.println("Se intinde covorul rosu");
    }
}
```

Spre deosebire de cazul constructorilor, apelul către varianta din parinte poate fi plasat oriunde în cadrul metodei subclasei (nu mai este obligat să se afle pe prima linie).

5.2.4.3.3. Clasa Object: overriding-ul este chiar recomandabil

Orice clasa Java este automat derivată din clasa Object fără ca programatorul să specifică acest lucru folosind `extends`, și chiar dacă clasa extinde o altă. Clasa Object reprezintă varful ierarhiei tuturor claselor din Java; ea conține proprietățile și metodele generale ale oricărui obiect. Implicit, toate obiectele Java mostenesc metodele publice ale acestei clase. Implementarea pe care aceste metode o au în clasa Object este puțin probabil să convina în subclase, și de aceea ea trebuie în general rescrisă.

Prezentăm trei dintre metodele clasei Object:

- **public String `toString()`** – returnează o reprezentare textuală a obiectului sub formă de String. Implementarea originală din clasa *Object* produce un sir de forma *tipDeDate@hashCode*, însă prin rescrierea metodei putem produce orice alte informații pe care le considerăm relevante despre obiectul în cauză. De exemplu, în cazul clasei *Caine*, putem rescrie metoda *toString()* astfel încât să afiseze direct numele patrupedului (vezi exemplul de mai jos).

Motivul pentru care am rescrie *toString()* și nu am crea propria noastră metodă de afisare a datelor obiectului este că *toString()* este deja utilizat într-un număr mare de metode ale diverselor clase din JRE, care se bazează pe corecta să rescriere. Exemplul notabil este cel al celebrei metode *System.out.println()* care, atunci când primește ca argument o referință către un obiect, îl apelează acestuia metoda *toString()* pentru a obține reprezentarea sa text:

```
class Caine {
    // ...diversi membri ai clasei, printre care numele
    public String toString() { return nume; }
    public static void main(String[] args){
        Caine c=new Caine("Azor");
        System.out.println(c); // Azor, în locul unui sir de forma Caine@87cfb2
    }
}
```

- **public boolean `equals(Object o)`** – realizează compararea (din punct de vedere al continutului!) a obiectului curent cu cel specificat ca parametru, și returnează **true** în cazul egalității. Implementarea originală din clasa *Object* efectuează comparare de referință (la fel ca operatorul ==); putem dicta regulile de comparare rescriind metoda *equals()*. De exemplu, pentru clasa *Persoana*, putem considera că două persoane sunt unul și același om când CNP-urile sunt identice:

```

class Persoana {
    // ...diversi membri, printre care CNP-ul
    public String toString() { return "Nume: "+nume+"\nCNP: "+CNP; }
    public boolean equals(Object o) { return CNP == ((Persoana)o).CNP; }
}

```

Pe aceasta metoda se bazeaza de asemenea multe facilitati Java, de remarcat fiind infrastructura de colectii (vezi capitolul corespunzator).

- **public int hashCode()** – aceasta metoda produce un numar calculat pe baza continutului obiectului. Numarul nu este unic de la un obiect la altul, insa trebuie sa fie acelasi pentru doua obiecte care sunt egale din punct de vedere al lui *equals()*. Programatorul trebuie sa aiba grijă sa pastreze implementarea lui *hashCode()* sincronizata cu cea a lui *equals()*: daca cea de-a doua isi schimba regulile de comparare, trebuie modificația și prima în consecință (explicații suplimentare vor fi oferite la capitolul dedicat colectiilor, la secțiunea despre implementările bazate pe hash table)

5.2.4.3.4. Relatia overloading-overriding (supraincarcare-rescriere)

Cele două procedee nu trebuie confundate; iată o prezentare comparativă:

Caracteristica comparată	OVERLOADING	OVERRDING
locatia metodelor	ambele în aceeași clasă; pot exista mai multe metode supraincarcate	originalul în clasa parinte, metoda rescrisă în clasa derivată; corespunzător ei există o singură metodă rescrisă în fiecare subclasa
nume	identic pentru toate metodele	identic pentru toate metodele
set de argumente	diferit de la o metodă la alta	identic pentru toate metodele
tip de date de întoarcere	poate fi diferit de la o metodă la alta	identic pentru toate metodele
nivel de acces	poate fi diferit de la o metodă la alta	metoda rescrisă nu poate avea nivel de acces mai restricтив

In concluzie, **supraincarcarea** ne da posibilitatea definirii a **mai multe metode diferite** dar apelabile cu **acelasi nume**, pe cand **rescrierea** ofera **implementari diferite** pentru **aceeasi metoda** mostenita in toate clasele dintr-o ierarhie.

5.2.5. Upcasting, downcasting si polimorfism

Dupa cum a fost prezentat anterior, relatia “is a” creata de mecanismul de mostenire ofera posibilitatea referirii unui obiect de subclasa prin intermediul unei referinte parinte:

```
Animal a = new Pisica();
```

Procedeul poarta denumirea de **upcasting** si este asemanator cu promovarea numerica din cazul primitivelor; atunci cand scriem `long L = 54;`, tipurile de date stang si drept nu sunt identice, insa compilatorul tine cont de faptul ca orice valoare de tip int poate fi cuprinsa intr-un long si efectueaza automat “largirea” tipului de date al valorii drepte la long. In acelasi mod, compilatorul sesizeaza relatia “is a” dintre Animal si Pisica si are astfel garantia ca obiectul referit va avea toate capabilitatile unui animal. Exista insa o diferență importantă în upcasting față de cazul primitivelor: **nu valoarea este cea convertita, ci referinta! Obiectul referit ramane permanent acelasi.**

Nota: am fi putut efectua un cast explicit: Animal a = (Animal) new Pisica();, insa acesta nu este necesar.

Upcasting-ul este prezent si in alte situatii, poate mai putin evidente; iata doua exemple:

- atunci cand o metoda primește un argument de tip Animal si ii pasam la apelare o referinta de tip Pisica. In interiorul metodelui va fi utilizata referinta de tip Animal prezenta in definitia metodelui
- atunci cand cream un tablou de tip Animal si il populam cu elemente de tip Pisica, Caine, Hamster etc. Fiecare element al tabloului va fi o referinta de tip Animal

Upcasting-ul introduce limitari, in sensul in care obiectul referit nu isi va putea manifesta in toate privintele “personalitatea” prin intermediul referintei parinte. Fie urmatorul exemplu:

```

class Animal{
    public void mananca(){}
}
class Vultur extends Animal{
    public void zboara(){}
    public static void main(String[ args){
        Animal a = new Animal();
        Vultur v = new Vultur();
        Animal x = new Vultur();      // upcasting
        a.mananca();                // ok, metoda proprie
        v.zboara();                  // ok, metoda proprie
        v.mananca();                // ok, metoda mostenita
        x.mananca();                // ok, metoda mostenita
        x.zboara();                 // EROARE, desi obiectul referit este de tip Vultur!
        Animal y = v;              // v a putut zbura; y refera acum acelasi obiect. Suspans...
        y.zboara();                 // ...y nu poate zbura, desi obiectul referit este acelasi!
    }
}

```

Referinta de tip parinte actioneaza ca un factor limitator: referintelor x si y din exemplu nu li se vor putea accesa decat membrii prezenti in clasa parinte, desi obiectele referite dispun cu certitudine de metoda suplimentara *zboara()*. Deducem urmatoarea regula: **setul de membri ce ii pot fi accesati unei referinte este determinat de tipul de date al referintei, indiferent de tipul de date al obiectului referit.**

Cum facem totusi ca vulturul referit de x sa zboare? Transformam referinta intr-o potrivita:

```

Animal x = new Vultur();
x.zboara();           // nu merge din cauza tipului de date al lui x
Vultur w = (Vultur) x; // se converteste referinta, nu obiectul!
w.zboara();           // ok, tipul de date al referintei o permite

```

O analogie utila este sa privim referinta ca fiind o telecomanda a obiectului. Atunci cand folosim un televizor cu telecomanda sa originala, avem acces la toate functiile sale; daca suntem nevoiti sa o inlocuim cu una universală, vom putea utiliza de la distanta doar functiile televizorului nostru care au corespondent in butoanele de pe telecomanda. Recunoastem in aceasta analogie referinta parinte (telecomanda universală) si referinta de tip subclasa (telecomanda originală).

Upcasting-ul isca o noua problema: nu putem accesa metodele suplimentare din subclasa, insa le putem accesa pe cele mostenite din parinte; ce se intampla daca subclasa rescrise metoda parinte? Care dintre implementari va fi folosita? Fie urmatorul exemplu:

```

class Om{
    public void vorbeste(){ System.out.println("Omul vorbeste"); }
}
class IlieDobre extends Om{
    public void vorbeste(){ System.out.println("Omul vorbeste cu o viteza rar intalnita"); }
    public static void main(String[] args){
        Om o1 = new Om();
        Om o2 = new IlieDobre();
        o1.vorbeste();      // implementarea din Om
        o2.vorbeste();     // care dintre implementari va fi folosita?
    }
}

```

Varianta folosita va fi cea din subclasa; posibilitatea ca doua apeluri diferite ale aceliasi metoda (*vorbeste()* din metoda main) realizate printr-o referinta de acelasi tip de date sa aiba efecte diferite in momentul executiei in functie de tipul *obiectului* referit poarta numele de **polimorfism**. Acest lucru este posibil datorita faptului ca metodele ce se apeleaza pentru o variabila de tip referinta nu sunt stabilite la compilare, in functie de tipul de date al referintei, ci in timpul rularii programului, abia in momentul apelului. Desi tipul de date al lui o2 este Om, , atunci cand apelam *o2.vorbeste()*

implementarea folosita va fi cea a clasei IlieDobre deoarece masina virtuala tine cont de tipul de date al obiectului referit de o2.

Stabilirea tarzie (in timpul rularii programului) a codului ce trebuie invocat corespunzator unui apel de metoda poate fi intalnita sub numele de *late binding*, *dinamic method invocation* sau *virtual method invocation*.

5.2.6. Mostenirea si membrii statici

Mostenirea actioneaza si pentru membrii statici, insa cu unele diferente fata de cazul membrilor de instanta:

- asemanari:
 - membrii statici se mostenesc
 - un membru static poate fi redeclarat
 - un camp static redeclarat in subclasa ascunde (mascheaza) campul din clasa parinte
 - o metoda statica poate fi rescrisa in subclasa
- deosebiri
 - in implementarea metodelor statice rescrise in subclase nu se poate folosi *super* pentru a refolosi codul parinte; functioneaza insa varianta *NumeClasaParinte.metodaStatica()*!
 - in cazul metodelor statice nu actioneaza polimorfismul! (vezi exemplul de mai jos)

```
class A{
    public static int x = 8;
    public static void f(){ System.out.println("Implementare parinte"); }
}
class B extends A{
    public static void f(){ System.out.println("Implementare subclasa"); }
}
class C extends A{
    public static boolean x = false;
}
class D extends B{
    public static void main(String[] args){
        A a1 = new A();
        B b1 = new B();
        A b2 = new B();
        C.f();           // Implementare parinte → metoda statica din A s-a mostenit
        D.f();           // Implementare subclasa → metoda statica din B s-a mostenit
        a1.f();          // Implementare parinte
        b1.f();          // Implementare subclasa
        b2.f();          // Implementare parinte → !!! nu functioneaza polimorfismul !!!
        System.out.println(A.x);   // 8
        System.out.println(B.x);   // 8 → mostenit din A
        System.out.println(C.x);   // false → in C a fost mascat campul original din A
    }
}
```

5.2.7. Interzicerea mostenirii

Java ofera programatorului posibilitatea sa "bata in cuie" definitia unei clase sau a unei metode, folosind cuvantul cheie final, dupa cum urmeaza:

- final aplicat unei metode interzice rescrierea acesteia in subclase. Subclasele mostenesc metoda in cauza insa nu o mai pot modifica
- final aplicat unei clase interzice extinderea acesteia. Acest lucru este util pentru clasele "strategice" in cazul carora existenta unor subtipuri de date ar crea probleme (un exemplu celebru este clasa String, declarata ca final)

5.3. Clase si metode abstracte: instantierea interzisa, implementarea obligatorie in subclase

Există numeroase cazuri în care dorim ca o clasa parinte să dispună de una sau mai multe metode pe care să le transmită descendenților sau însă clasa este prea generală pentru a oferi o implementare metodelor, sau chiar pentru a fi instantiată. De exemplu, în clasa *Animal*, implementarea metodei *scoateSunet()* nu-si gaseste sensul (cum face un animal “generic”?), în schimb ea este o manifestare a tuturor animalelor și deci trebuie să fie prezenta în clasa *Animal*; metoda va fi apoi particularizată pentru fiecare tip de animal în parte, în subclase (cainele latra, pisica miauna etc). În plus, realizăm ca clasa *Animal* nici nu are sens să fie instantiată – aplicația va folosi instante ale subclaselor. Suntem asadar în situația în care dorim două lucruri aparent contradictorii:

- clasa parinte trebuie să aibă o metodă pe care să o transmită descendenților
- nu dorim să oferim implementare metodei în clasa parinte, dar în schimb dorim să forțăm subclaselor să ofere fiecare propria implementare

Java ne dă posibilitatea de a defini metode care nu au implementare, ci sunt folosite doar ca place-holder, urmand să primească implementare în obiectele derivate. Aceste metode se numesc **abstracte** și sunt precedate la declarare de calificatorul **abstract**:

```
abstract class Animal {
    public abstract void scoateSunet(); // nu mai există corp al metodei !
    public void seHraneste() { ... }
}
```

Orice clasa care are cel putin o metoda abstracta trebuie la randul sau declarata ca abstract. O astfel de clasa are o definitie incompleta (lipseste implementarea uneia sau mai multor metode) și de aceea nu se poate instantia.

Nota: o clasa poate fi declarata ca abstract si fara sa contina metode abstracte.

Proprietatile unei clase abstracte sunt:

- nu se poate instantia; este doar un model ale căruia caracteristici sunt preluate în subclase
- poate cuprinde atât metode abstracte cât și concrete
- metodele ei abstracte nu au implementare – ele reprezintă doar o obligație pentru subclase
- nu are constructor, în scopul apelării lui din subclase
- orice subclasa a unei clase abstracte este obligată să adopte una din două variante:
 - dacă este clasa concretă, ea trebuie să rescrie (override) toate metodele abstracte moștenite din clasa parinte
 - poate fi declarată la randul ei ca *abstract*, condiții în care nu mai are nicio obligație

Asadar, prin utilizarea unei clase abstracte, forțăm subclaselor să ofere propria implementare metodelor abstracte moștenite din parinte.

Nota: calificatorul abstract nu poate coexista cu final în cadrul aceleiasi definitii de metoda deoarece au efecte contradictorii: abstract impune rescrierea metodei, iar final o interzice.

5.4. Interfete: iata forma, implementati continutul

Există multe tipuri de obiecte – atât în viața reală cât și în programare – care au capabilități comune fără a fi de tipuri de date înrudite (cel puțin, nu în sensul de inheritance). Neexistând relația “*is a*” între cele două tipuri de date sau o clasa parinte comună, capabilitatea comună ale celor două tipuri de date nu ar putea fi refolosită eficient. În tot ceea ce se prezintă până acum în acest material relațiile dintre clase erau unele de inclusiune totală: un Mamifer “*is a*” *Animal*, deci și se vor putea apela toate metodele public disponibile ale clasei *Animal*, ceea ce ne permite să folosim un obiect Mamifer oriunde este nevoie de unul *Animal*. Există însă o multitudine de animale care mananca paine (spre exemplu) fără a se afla pe aceeași verticală a ierarhiei de clase; cum refolosim această capabilitate?

Pentru a intelege mai bine problema, sa adaugam cateva exemple din viata reala:

- pentru a sparge un ou putem folosi orice obiect tare, indiferent de natura sa. Capabilitatea de a sparge obiecte casante este comună multor obiecte altminteri distințe
- daca consideram o ierarhie de clase ce descrie diversele profesii umane, constatam ca diferite profesii din ierarhie presupun lucru stand pe scaun pe perioade lungi fara a se afla in vreo relatie una cu alta. Aceste oameni au in comun actiuni precum ridicarea de pe scaun sau asezarea, plus complicatiile de sanatate atasate sedentarismului (care s-ar materializa in metode comune tuturor acelor clase)
- data fiind o ierarhie de clase corespunzatoare genurilor de filme, constatam ca ele pot fi clasificate din mai multe puncte de vedere: pe de o parte din punct de vedere al genului (drama, actiune, horror, comedie, animatie etc) dar si din alte puncte de vedere (artistice, serial; scurt metraj, lung metraj). Daca construim o ierarhie de clase bazata pe durata, constatam ca multe subclase (filme concrete) de lung sau scurt metraj se vor incadra in anumite categorii (animatie, drama etc) - si deseori in mai multe simultan - si vom dori sa definim de asa natura caracteristicile lor comune incat sa le putem refolesi. De fapt, dorim crearea de categorii care sa depaseasca granitetele ierarhiilor de clase - niste categorii trans-ierarhie.
- ducand paralela mai departe: daca avem in fata noastră un grup de oameni de diferite profesii, ne putem adresa ad-hoc unei categorii (unui subgrup) cu o fraza de genul: "schiorii, un pas in fatal! puneti claparii si executati trei viraje!" In acest fel am "adunat" si am "pus la treaba" obiecte diferite ca tipuri de date (ingineri, constructori, medici etc) dar avand toate un mic set de capabilitati comune - iar noi ne-am adresat tocmai acelor capabilitati: am creat un tip de date trans-ierarhie profesionala ("schiorii") si am apelat tuturor obiectelor metode pe care avem certitudinea ca le au: *puneClapari()* si *executaViraj()*.
- o alta analogie: certificările. Persoanele care se certifica Java (si care pot face parte din orice ierarhie, din diverse puncte de vedere) au un anumit set minimal de capabilitati. Certificarea pe care o obtin gestioneaza interacțiunea intre ei si angajatorii: o persoana certificata are un set de capabilitati binecunoscut, iar angajatorul ii va "apela" persoanei "metodelor" corespunzatoare (pe care este sigur ca le are) odata angajarea realizata. Certificarea este deci o forma de reglementare a intalnirii dintre cele doua entitati, o interfata.

Corespondentul tipurilor de date trans-ierarhice prezentate in exemplele de mai sus este interfata Java. Ea reprezinta un tip de date care descrie un set de capabilitati comun unei categorii trans-ierarhice de obiecte. Interfata include metode insa niciuna nu are implementare - toate metodelor sale sunt abstracte. Interfata era pura forma, fara continut. In consecinta, ea poate fi privita ca un fel de contract sau de standard: toate clasele care adera la standard sunt obligate sa rescrie (de fapt sa ofere implementare pentru) metodelor abstracte mostenite din interfata, si deci aderarea la standard aduce cu sine garantia prezentei unui set de metode in API-ul clasei. Interfata este "certificarea" unei clase - cea care garanteaza prezența unui anumit set de metode.

Nota: interfetele sunt o entitate distinctă a mașinii virtuale Java, cu regim diferit față de primitive, clase, tablouri și enum-uri.

Tehnic, o interfata Java se supune urmatoarelor reguli:

- se declara folosind cuvantul cheie **interface** (si nu class ca pana acum!)
- nu se poate instantia
- poate contine campuri si metode, dar cu urmatoarele restrictii:
 - campurile sunt automat public, static si final, chiar daca programatorul nu scrie acesti modificatori si calificatori in definitia atributului! Nu exista campuri de instanta intr-o interfata
 - metodelor sunt automat public si abstract, chiar daca programatorul nu specifica acest lucru
- nu are constructor
- o clasa concreta sau abstracta nu extinde, ci implementeaza o interfata, folosind in acest scop cuvantul cheie **implements**
- o clasa concreta sau abstracta poate implementa mai multe interfete simultan, colectand astfel capabilitati din mai multe surse (ex: o persoana cu profesia de inginer poate fi in acelasi timp schior si montagnard).
- o interfata poate **extinde** una sau mai multe alte interfete (folosind cuvantul cheie **extends**, nu *implements*)
- implementarea unei interfete creeaza tot o relatie "is a":

```
class Titanic extends FilmArtistic implements Drama, Dragoste{}  
// un obiect de tip Titanic este un Drama!
```

Iata un exemplu ce contine o ierarhie mai complexa de clase si interfete: avem cateva interfete care descriu capabilitati ale obiectelor - *ObiectTare*, *ObiectGreu*, *ObiectDeLovit*, *Descriptibil* - si clase concrete care implementeaza in diverse combinatii sau folosesc intern aceste interfete: *Piatra*, *Laptop*, *TeancHartii*.

```

interface Descriptibil{ String descriere(); }
interface ObiectTare{}
interface ObiectGreu{}
interface ObiectDeLovit extends ObiectTare,ObiectGreu{
    void loveste();
}
interface ObiectCasant extends Descriptibil{
    void spargeCu(ObiectDeLovit o);
}

class Ou implements ObiectCasant{
    public void spargeCu(ObiectDeLovit o){
        o.loveste();
        System.out.print(descriere()+" a fost facut chisalita cu ");
        if(o instanceof Descriptibil)
            System.out.println((Descriptibil)o.descriere()); // downcast!
        else
            System.out.println(" un obiect tare si greu");
    }
    public void descriere(){ return "Oul Humpty Dumpty"; }
}
class Laptop implements ObiectDeLovit,Descriptibil{
    String model;
    public Laptop(String m) { model = m; }
    public void loveste(){
        System.out.println("Laptopul se ridica amenintator si cazu cu un zgomot infernal");
    }
    public void descriere(){ return "un laptop "+model; }
    public void porneste(){ System.out.println("Laptopul booteaza"); }
}
class Piatra implements ObiectDeLovit{
    public void loveste(){
        System.out.println("Piatra despica aerul si lovi puternic");
    }
}
// pot folosi un ObiectGreu si pentru a tine foi, nu doar pentru a sparge
class TeancHartii{
    void propteste(ObiectGreu o){
        System.out.println("Hartiile nu mai zboara daca se face curent");
    }
}
public class Main{
    public static void main(String[] args){
        Laptop L = new Laptop("HP");
        ObiectDeLovit OL = new Laptop("Asus");
        Ou o = new ou();
        Piatra p = new Piatra();
        TeancHartii t = new TeancHartii();
        o.spargeCu(L);
        o.spargeCu(p);
        t.propteste(L); // un Laptop "is a" ObiectDelovit care "is a" ObiectGreu
    }
}

```

Un obiect a carui clasa implementeaza o interfata poate fi accesat folosind o referinta cu tipul de date al interfetei, insa i se vor putea acces numai metodele care tin de interfata: in metoda *main()* de mai sus, lui *L* i se pot apela toate metodele din cadrul clasei *Laptop*, pe cand lui *OL* i se pot apela doar metodele *loveste()* si *descriere()*!

Interfetele sunt folosite pe larg in cadrul JRE - iata cateva exemple:

- interfata *CharSequence* care e implementata de clasele *String*, *StringBuffer* si *StringBuilder*, care deriva toate direct din *Object* (nu au relatii directe intre ele). Orice obiect *CharSequence* ofera garantia posibilitatii de accesare a caracterului de pe o anumita pozitie, de extragere a lungimii, reprezentare text etc., ceea ce se

materializeaza in lista de metode a obiectelor (*charAt()*, *length()*, *subSequence()*, *toString()*). Am putea crea o metoda de validare de nume de persoana care primeste ca argument CharSequence si verifica ca toate caracterele sunt de tip litera:

```
boolean valideazaNumePersoana(CharSequence c) {
    for(int i=0; i< c.length(); i++) {
        if (!Character.isLetter(c.charAt(i))) {
            return false;
        }
    }
}
```

Acestei metode ii putem pasa acum orice argument de tip String, StringBuffer sau StringBuilder.

- interfata *Comparable*. Orice clasa care implementeaza *Comparable* va avea o metoda *compareTo()*; algoritmii de comparare deja prezenti in unele clase Java impun ca obiectele comparate sa fie “is a” *Comparable*, prin aceasta avand garantia ca dispun de metoda de comparare *compareTo()*
- interfetele colectie. Spre exemplu, orice clasa colectie care implementeaza interfata *List* va dispune de metode pentru adaugare si stergere de obiecte din colectie, identificarea pozitiei unui obiect, extragerea obiectului de pe o anumita pozitie etc.

In concluzie, o interfata realizeaza standardizarea interactiunii unei categorii trans-iererhice de obiecte cu exteriorul, oferind codului client garantii.

5.5. Pachete java (packages)

5.5.1. Principii

Un pachet Java reprezinta un grup de clase ce slujesc unui scop comun aflate sub umbrela aceliasi nume. Apartenenta unei clase la un pachet ofera urmatoarele avantaje programatorului:

- o mai buna organizare a claselor proiectului. Spre exemplu, clasele care tin de interfata grafica pot fi plasate intr-un pachet numit *gui*, cele care tin de manipularea datelor intr-unul *model* etc
- rezolvarea conflictelor de nume. Este posibil ca in acelasi proiect sa existe clase provenite de la programatori diferiti si care au fost denumite identic. In lipsa pachetelor, aceste clase nu pot coexista
- crearea unor relatii mai stranse intre anumite clase. Un membru al unei clase (camp/metoda) cu nivel de acces *default* este accesibil pentru clasele din acelasi pachet si invizibil pentru cele din afara pachetului

5.5.2. Declararea apartenentei la un pachet

Apartenenta unei clase la un anumit pachet se stabileste folosind declaratia *package* la inceputul fisierului sursa:

```
// fisierul OpenDialog.java
package gui;
public class OpenDialog { /*...corful clasei...*/ }
```

```
// fisierul ImageChooser.java
package gui.imageprocessing;
public class ImageChooser{ /*...corful clasei...*/ }
```

Atentie! Declaratia package trebuie sa fie prima instructiune necomentata din fisierul sursa! (inaintea ei sunt admise numai linii de comentariu).

Toate clasele care nu beneficiaza de declaratie *package* fac parte din asa-numitul pachet default, asadar sunt considerate a face parte din acelasi pachet. Aceasta practica este insa descurajata deoarece anuleaza o parte dintre avantajele sistemului de pachete.

5.5.3. Numele pachetelor

Numele unui pachet poate fi:

- un nume simplu, de forma *gui* sau *resources*. Aceasta constă dintr-un singur identificator, care poate folosi litere, cifre, underscore. Se recomandă ca literele folosite să fie mici, pentru a nu fi confundat numele de pachet cu un nume de clasa sau variabilă
- un nume compus, de forma *gui.filemanagement* sau *javax.swing.table*. Un astfel de nume constă dintr-o succesiune de nume simple separate prin punct.

Numele compuse se folosesc în două scopuri:

- pentru a sugera relații între pachete. Spre exemplu, *javax.swing* și *javax.swing.table* sunt două pachete care contin clase folosite în interfețe grafice Swing; *javax.swing* conține clase mai generale, iar *javax.swing.table* conține clase legate de componenta JTable. **Atenție însă! Desi *javax.swing.table* pare să fie un subpachet al lui *javax.swing*, cele două pachete nu se contin unul pe altul și nu se află într-o relație de inheritance, ci reprezintă pachete distincte!** (în consecință, precum se va vedea, solicită și directive *import* separate)
- pentru a minimiza probabilitatea coliziunilor de nume. Se recomandă ca firmele care dezvoltă soft să folosească nume de pachete ce pornesc de la domeniul lor DNS, după cum urmează: dacă domeniul internet al firmei este firma.com, atunci toate pachetele scrise de programatorii acelei firme vor avea prefixul com.firma (ex: com.firma.gui, com.firma.resources.images etc). Această soluție nu face decât să se folosească de o infrastructură de nume unice deja existentă (cea DNS) pentru a nu crea una proprie

5.5.4. Accesarea claselor ce fac parte dintr-un pachet

5.5.4.1. Implicații ale apartenenței la pachet

Când o clasa face parte dintr-un pachet, numele său devine unul mai lung, format prin prefixarea numelui clasei cu numele pachetului:

```
gui.OpenDialog d = new gui.OpenDialog.Unu();
```

Acest mod de denumire rezolvă problema coliziunii numelor (putem distinge clasa OpenDialog de o altă cu același nume însă aflată în alt pachet). Cele două clase pot astfel coexista în același proiect. Este același lucru ca în cazul sistemului de fisiere – putem avea două fisiere numite *File1* însă aflate în două directoare diferite, diferenția între ele facându-se cu ajutorul caii absolute către fiecare fisier.

Instructiunea *package* influențează vizibilitatea inter-clase: o clasa declarată cu nivel de acces default este disponibilă (vizibilă) pentru toate celelalte clase din acel pachet, și invizibilă din afara. Dacă dorim ca acel tip de date să fie folosit și în clase din afara pachetului va trebui să declarăm clasa noastră cu nivel de acces public. O clasa în care nu s-a specificat instructiunea *package* face parte din pachetul default, adică va fi vizibilă pentru toate clasele aflate în același director și care nu au declarat *package*.

Accesarea unei clase care face parte dintr-un pachet se realizează diferențiat, în funcție de relația între clasa care accesează și cea accesată:

- dacă ambele clase fac parte din același pachet, atunci se pot referi una la cealaltă folosind numele lor scurt:

```
package gui;
class OpenDialog{}
```

```
package gui;
class MainWindow{
    OpenDialog d;
}
```

- daca cele doua clase fac parte din pachete diferite, una nu o poate accesa pe cealalta decat daca cea de-a doua are nivel de acces public in pachetul sau, si numai folosind numele ei lung:

```
package animale;
class Pasare{}
```

```
package wildlife;
class Stol{
    animale.Pasare[] pasari = new animale.Pasare[10];
}
```

5.5.4.2. Utilizarea directivei import

5.5.4.2.1. Principii. Forme ale directivei

O clasa poate fi referita de numeroase ori in cadrul surselor unui proiect; daca ea este accesata din afara pachetului său, este necesara de fiecare data utilizarea numelui complet al clasei (*numepachet.numeclasa*). Acest lucru devine mai incomod pe masura ce lungimea numelui de pachet creste si, in plus, dauneaza lizibilitatii codului. De aceea au fost luate masuri pentru ca programatorul sa poata folosi, pe cat posibil, numele scurt al clasei.

Atunci cand dorim sa accesam, din cadrul unui pachet, clase ce fac parte dintr-un alt pachet (cum este cazul tuturor claselor ce fac parte din JRE), Java ne pune la dispozitie o modalitate de a specifica de la bun inceput la ce clasa ne referim, sau altfel spus in ce ierarhie de pachete va fi cautata o clasa referita cu numele ei scurt: directiva import, care “face vizibile” clase din pachete diferite. Exista doua forme ale directivei:

- forma care refera o clasa exacta (ex: import gui.openDialog). Este folosita pentru clase punctuale
- forma care refera toate clasele unui pachet (ex: import java.util.*). Este utila atunci cand se folosesc mai multe clase din cadrul aceluiasi pachet si, deci, nu ar fi practica importarea fiecareia in parte

5.5.4.2.2. Importarea unei clase punctuale

Sa consideram exemplul:

```
package animale.mici;
class Purice{}
```

```
package animale.medii;
public class Catzel {
    Purice p=new Purice();
}
```

La incercarea de compilare a clasei *Catzel* compilatorul ar genera o eroare, deoarece nu exista o clasa *Purice* in pachetul *animale.medii*. Solutia este sa-i indicam compilatorului numele complet al clasei *Purice*, astfel incat, la orice aparitie a numelui scurt al clasei, acesta sa stie la ce clasa ne referim:

```
import animale.mici.Purice; //ACEA clasa Purice
public class Catzel {
    // acum nu mai este nevoie sa scriem decat numele scurt al clasei
    Purice p=new Purice();
}
```

Locul directivei import este inaintea declaratiei clasei si dupa declaratia *package*. Acest lucru NU implica insa faptul ca clasa *Purice* va fi incarcata automat la pornirea programului! Directiva import stabileste un simplu prefix – o indicatie catre compilator: ori de cate ori este referita clasa *Purice* cu numele sau scurt, va fi folosita clasa *animale.mici.Purice*.

5.5.4.2.3. Importarea tuturor claselor dintr-un pachet

Atunci cand folosim mai multe clase din cadrul aceluiasi pachet extern ar fi nevoie de importarea fiecareia in parte, ceea ce nu este nici eficient, nici practic (inceputul fisierului sursa ar fi poluat cu o lunga lista de directive import). Pentru a rezolva aceasta problema, directiva import suporta si o sintaxa care importa toate clasele unui pachet:

```
import animale.mici.*;
import animale.medii.*; // orice clasa referita cu numele scurt va fi cautata in aceste pachete
public class Catzel {
    Purice p = new Purice();
}
```

Prin acest procedeu se stabilesc prefixe posibile pentru numele scurte de clase ce apar in program. Putem avea mai multe astfel de directive import; la intalnirea unui nume scurt, compilatorul va incerca sa prefixeze pe rand numele clasei cu numele de pachete prezente in directivele import cu wildcard (*), pana cand gaseste o clasa valida.

Daca doua sau mai multe pachete importate prin acest procedeu contin clase cu nume identice, acest lucru nu reprezinta o problema (si nu se materializeaza intr-o eroare de compilare) decat atunci cand se incercă referirea clasei in cauza cu numele ei scurt; compilatorul va semnala prompt ambiguitatea. Rezolvarea consta din folosirea numelui complet al clasei.

Nota: urmatoarele pachete sunt importate automat: pachetul curent, pachetul default si pachetul java.lang.*.

Pentru orice clasa ce nu face parte dintr-unul dintre pachetele importate automat, programatorul are doua solutii:

- sa importe explicit clasa in cauza, folosind una dintre cele doua forme ale directivei import
- sa foloseasca numele complet al clasei

Atentie! Chiar daca un pachet pare a contine un sub-pachet, cele doua pachete sunt distincte si importarea pachetului "parinte" nu va importa automat si "sub-pachetul". Sunt necesare directive import separate:

```
import javax.swing.*;
import java.swing.text.*;
import javax.swing.text.html.*;

class DoNotPushThisButton extends JButton{...}
```

5.5.4.2.4. Importarea membrilor statici ai unei clase

In versiunile noi de Java a fost introdusa o alta forma de import – cea pentru campuri statice. Aceasta este utila atunci cand o clasa are membri statici publici des accesati, care ar solicita folosirea de fiecare data a numelui clasei (ex: Math.random(), Integer.MAX_VALUE). Pentru a putea folosi astfel de membri cu numele lor scurt exista directiva *import static*:

```
import static java.lang.Integer.*;
import static java.lang.Math.PI;

class A{
    void f(){
        System.out.println(parseInt("56"));           // metoda parseInt() folosita cu numele scurt
        System.out.println(PI);                         // constanta PI accesata cu numele scurt
    }
}
```

Import-ul static reprezinta o solutie comoda doar la prima vedere si trebuie folosita cu masura, deoarece poate face codul mai greu de urmarit. Spre exemplu, daca intr-un fisier avem mai multe directive *import static*, la o lectura ulterioara a codului nu vom sti din ce clasa provine o metoda/camp accesat cu numele scurt!

5.5.5. Structura de pachete JRE

Toate pachetele ce formeaza JRE au un nume ce incepe cu *java* sau *javax* si continua cu unul sau mai multe nume simple, in functie de scopul lor. Iata cateva exemple de pachete utile:

- **java.lang** – contine clasele “strategice” Java. Aici se gasesc String, StringBuffer, StringBuilder, Math, clasele de impachetare, clasa System etc. Pachetul are regim special, in sensul in care este importat automat, fara ca programatorul sa ia vreo masura in acest sens
- **java.util** – contine o multitudine de clase de utilitate generala, cum ar fi Calendar, Date, Random, clasele de tip colectie etc.
- **java.awt** – contine clasele corespunzatoare componentelor grafice AWT
- **javax.swing** – idem dar pentru componente Swing
- **java.io** – contine clase pentru lucrul cu fluxuri de date (citire din scriere in fisiere/conexiuni de retea) si serializare
- **java.sql** – contine clase folosite pentru interfatare cu servere de baze de date

5.5.6. Mecanismul de incarcare a claselor

In Java, fiecarui pachet ii corespunde un director propriu. Intrebarea este: unde trebuie sa se afle acest director si ce trebuie sa facem ca masina virtuala Java sa "vada" clasele din pachetul nostru?

Raspunsul vine sub forma lui CLASSPATH. Precum se stie, JVM incarca clasele dinamic, in momentul in care este nevoie de ele (la accesarea unui membru static public sau la crearea unei variabile de acel tip de date). Pentru a gasi definitia clasei respective, JVM va cauta fisierul .class corespunzator considerand ca radacina a ierarhiei de pachete directoarele specificate in CLASSPATH.

In figura 1 este prezentat un exemplu de configurare a variabilei CLASSPATH pentru Windows. Utilizatorul ii indica masinii virtuale sa caute clase in ierarhiiile de pachete aflate in directoarele d:\work si c:\java\clase. Cautarea nu se face recursiv - daca numele clasei este Trei, va fi cautat un fisier d:\work\Trei.class sau c:\java\clase\Trei.class, NU se va cobori recursiv in directoarele DIR1 sau DIR2.

Numele complet al clasei poate contine o insiruire de pachete si subpachete: de exemplu, DIR2.DIR3.Patr. Specificarea completa a numelui permite o localizare exacta a clasei pe care dorim s-o incarcam (desi exista clase Patru si in DIR2 si in DIR1). Modul de lucru este asemanator sistemului DNS din retele.

CLASSPATH-ul poate fi setat si dinamic:

- la compilare, putem folosi optiunea javac –classpath cale fisier.java
- la rulare, putem folosi java –cp cale fisier

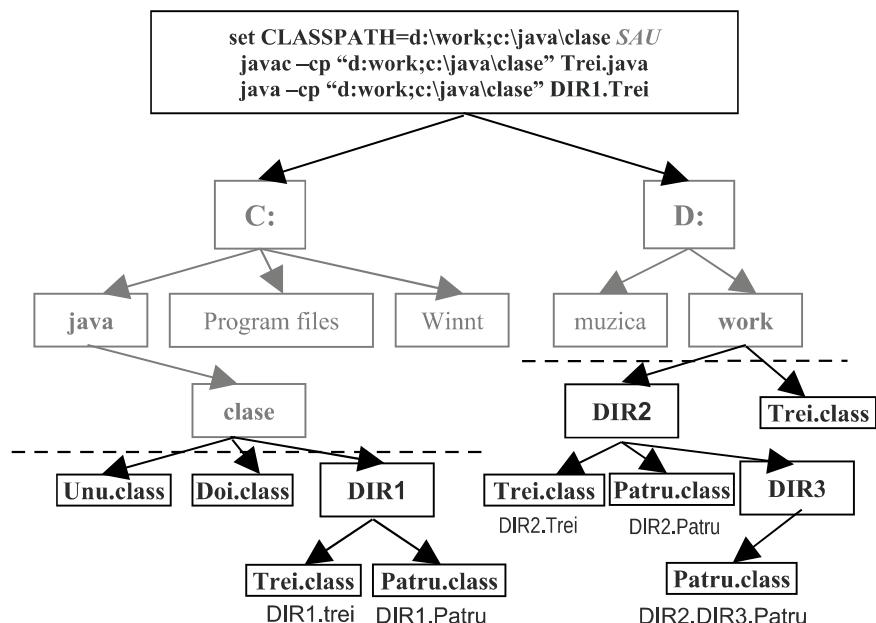


FIGURA 1

Este foarte important ca structura de directoare sa corespunda cu numele complet al pachetului; cand intalneste instructiunea *import d2.d3.Patr*, JVM va transforma toate caracterele “.” in „/” (sau „/” pentru Unix/Linux) si va cauta fisierul *d2/d3/Patr.class* intr-una din caile specificate in CLASSPATH. In cazul nostru, JVM va cauta *d:\work\d2\d3\Patr.class* sau, daca acesta nu este gasit, *c:\java\clase\d2\d3\Patr.class*.

5.6. BIBLIOGRAFIE

- Inheritance:
 - <http://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html>
- Interfete Java:
 - <http://docs.oracle.com/javase/tutorial/java/concepts/interface.html>
 - <http://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html>
- Upcasting si downcasting: <http://forum.codecalle.net/java-tutorials/20719-upcasting-downcasting.html>