

Programare orientată pe obiecte

#13 JAVA
Mostenire

Adrian Runceanu
www.runceanu.ro/adrian

Curs 13

Moștenire



Moștenire

1. Conceptul de moștenire
2. Caracteristicile unei ierarhii de clase
3. Moștenirea simplă și multiplă
4. Controlul accesului și moștenirea
5. Metode-constructor pentru clase derivate și cuvântul-cheie super
6. Supraîncărcarea (overloading) metodelor
7. Redefinirea (override) metodelor
8. Redefinirea parțială a unei metode

1. Conceptul de moștenire

- Conceptul de mostenire este folosit in **Java** la organizarea claselor si a comportamentului acestora.
- Mostenirea este mecanismul fundamental pentru *refolosirea codului*.
- Prin mostenire *o clasa noua dobandeste toate attributele si comportamentul unei clase existente* (clasa originala).
- Din acest motiv, noua clasa poate fi creata prin simpla specificare a diferentelor fata de clasa originala din care provine.

1. Conceptul de moștenire

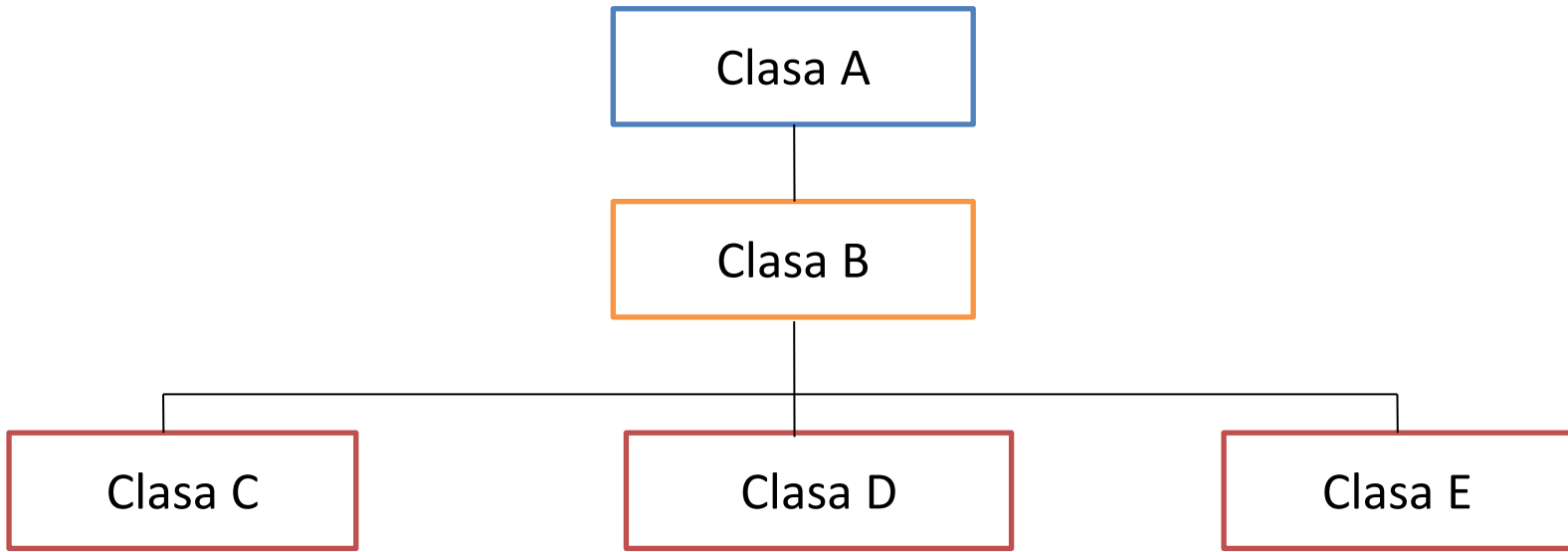
- Datorita relațiilor de moștenire clasele de obiecte sunt organizate într-o **ierarhie** bine precizată.
- În cadrul ierarhiei claselor de obiecte, **operația de definire a unei noi clase de obiecte** pe baza uneia deja existente se numește **derivare**.
- **Clasa originală** (mai generală) se va numi **superclasa** a noii clase, iar **noua clasă de obiecte** se va numi **subclasa** a clasei din care derivă.

1. Conceptul de moștenire

- Uneori, în loc de derivare se folosește termenul de **extindere**.
- Termenul vine de la faptul că o subclasă își extinde superclasa cu noi variabile și metode.
- De exemplu putem extinde clasa *Masina*:
 - la *MasinaStraina* (pentru care se plătește vama)
 - și *MasinaAutohtona* (pentru care nu se plătește vama)
etc.

1. Conceptul de moștenire

- Intr-o ierarhie de clase, **o clasa poate avea o singura superclasa**, insa poate avea *un numar nelimitat de subclase*.
- Subclasele mostenesc toate attributele si metodele superclasei lor, care la randul ei mosteneste toate attributele si metodele de la superclasa ei si asa mai departe, urcand in ierarhie.
- Rezulta ca *o clasa nou creata contine toate attributele si metodele claselor aflate deasupra sa in ierarhie, si in plus contine propriile attribute si metode*.



Clasa A este superclasa a clasei B

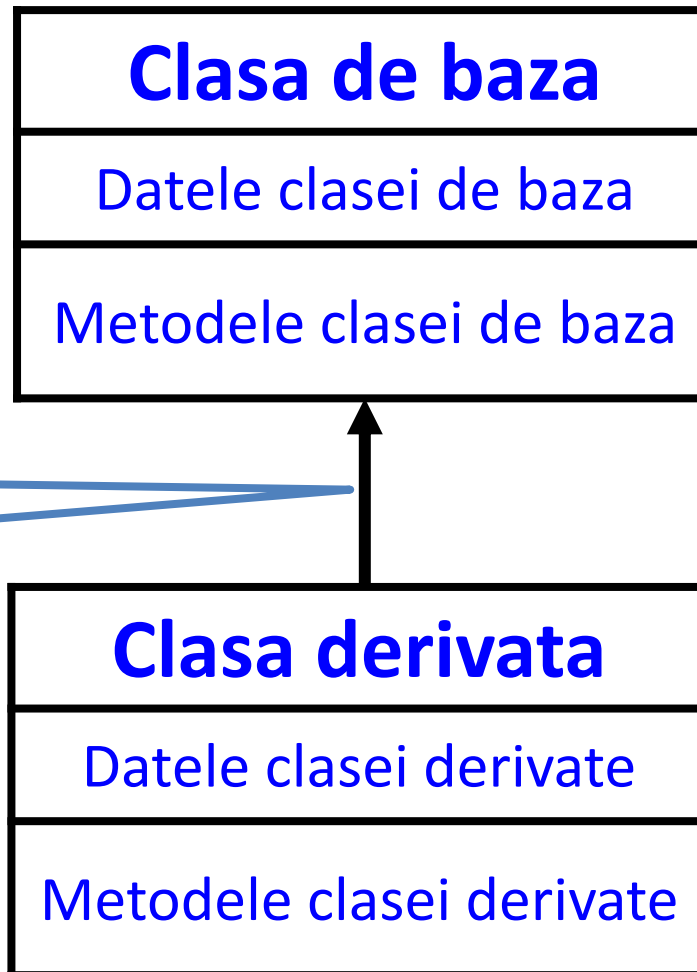
Clasa B este subclasa a clasei A

Clasa B este superclasa pentru clasele C, D si E

Clasele C, D si E sunt subclase ale clasei B

1. Conceptul de moștenire

Reprezentarea relatiei de mostenire folosind diagrama UML (Unified Modeling Language):

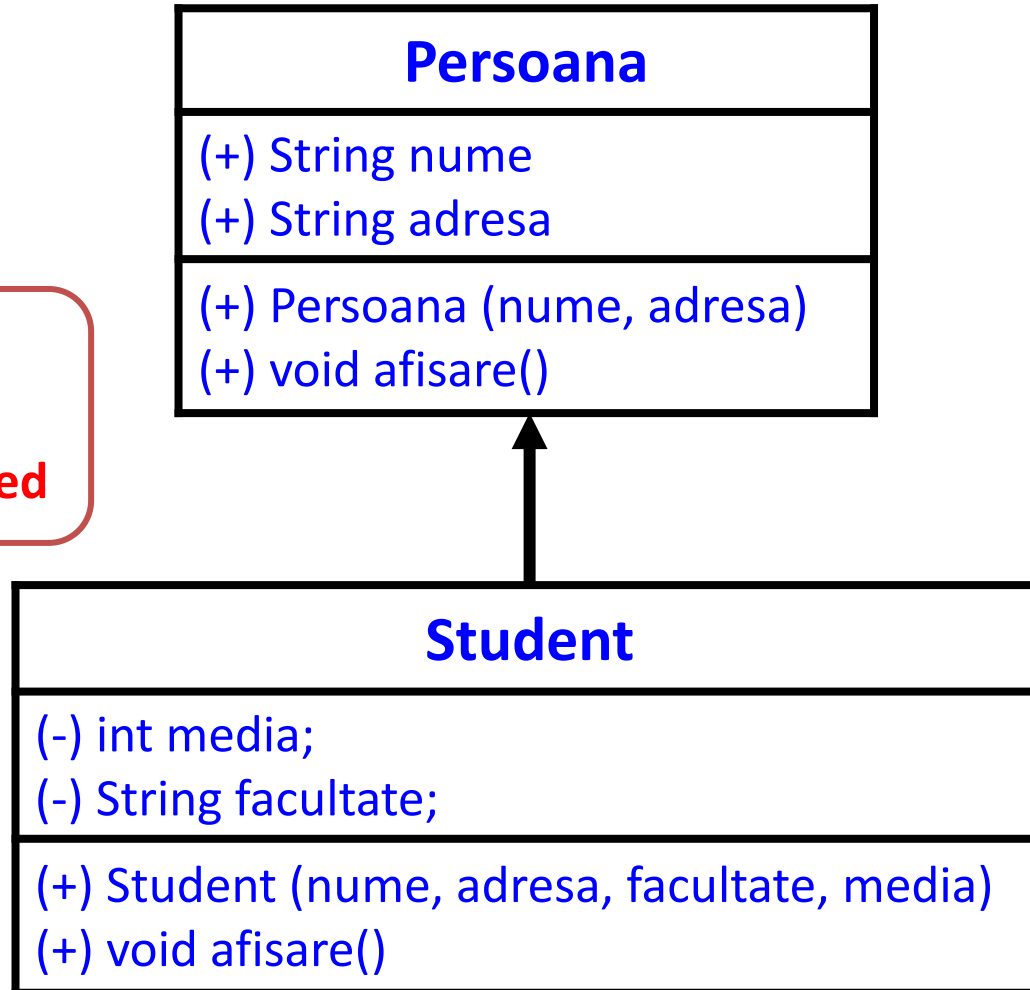


Relatia de mostenire

1. Conceptul de moștenire

Exemplu:

(+) semnifica **public**
(-) semnifica **private**
(#) semnifica **protected**



1. Conceptul de moștenire

- In varful ierarhiei de clase Java se afla **clasa *Object***, iar toate clasele de obiecte, care se creeaza, sunt derivate din aceasta unica superclasa.
- ***Object*** reprezinta *clasa initiala, sa-i spunem clasa de obiecte generice, care defineste attributele si comportamentul (metodele) mostenite de toate clasele din biblioteca de clase **Java**.*
- In varful ierarhiei se definesc concepte (clase) abstracte, foarte generale.
- Aceste concepte generale devin din ce in ce mai concrete, mai particularizate, o data cu “coborarea” spre subclasele de pe nivelele de mai jos in ierarhie.

1. Conceptul de moștenire

- Când clasa nou creată definește un comportament complet nou și nu este o subclasă a unei alte clase, atunci ea moștenește direct clasa **Object**.
- Astfel, noua clasă se integrează corect în ierarhia claselor **Java**.
- De asemenea, dacă se definește o clasă care nu specifică o superclasă, Java presupune că noua clasă moștenește direct clasa **Object**.
- Clasele definite în exemplele de până acum au moștenit direct clasa **Object**.

Mostenire

1. Conceptul de moștenire
2. Caracteristicile unei ierarhii de clase
3. Moștenirea simplă și multiplă
4. Controlul accesului și moștenirea
5. Metode-constructor pentru clase derivate și cuvântul-cheie super
6. Supraîncărcarea (overloading) metodelor
7. Redefinirea (override) metodelor
8. Redefinirea parțială a unei metode

2.Caracteristicile unei ierarhii de clase

- Organizarea unei aplicatii informatice intr-o ierarhie de clase presupune o planificare atenta.
- De alegerea claselor si de proiectarea arborelui acestor clase depinde eficienta si flexibilitatea aplicatiei.

2.Caracteristicile unei ierarhii de clase

Principalele *caracteristici ale unei ierarhii de clase* sunt:

- **atributele si metodele comune** mai multor clase pot fi definite in superclase, care permit folosirea repetata a acestora pentru toate clasele aflate pe nivelele mai joase din ierarhie
- **modificarile efectuate in superclasa** se reflecta automat in toate subclasele ei, subclasele acestora si asa mai departe; *nu trebuie modificat si recompilat nimic in clasele aflate pe nivelurile inferioare, deoarece acestea primesc noile informatii prin mostenire*

2.Characteristicile unei ierarhii de clase

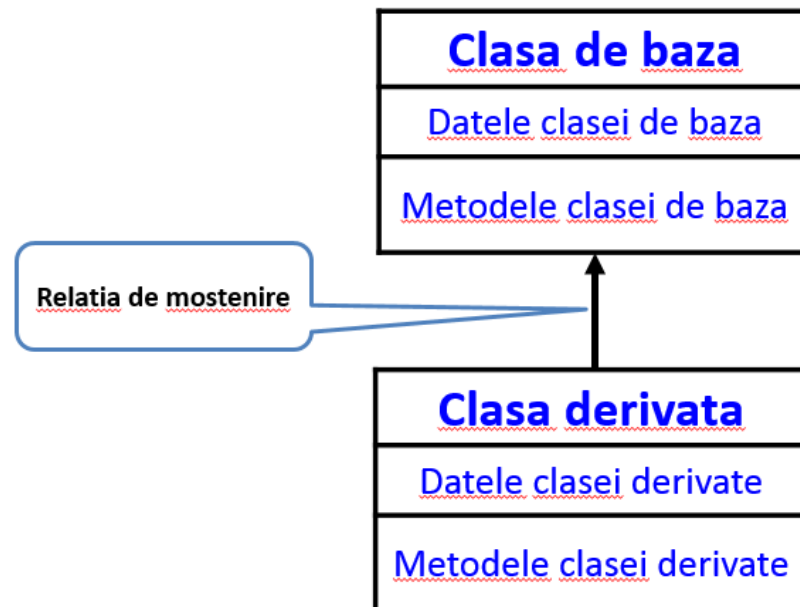
- **clasa** derivata poate sa adauge noi attribute si metode si poate **modifica semnificatia metodelor mostenite** (prin **polimorfism**);
- **superclasa** nu este afectata in nici un fel de modificarile aduse in clasele derivate;
- **o clasa derivata este compatibila ca tip cu superclasa**, ceea ce inseamna ca o variabila referinta de tipul superclasei poate referi un obiect al clasei derivate, dar nu si invers; clasele derivate dintr-o superclasa nu sunt compatibile ca tip.

Mostenire

1. Conceptul de moștenire
2. Caracteristicile unei ierarhii de clase
3. Moștenirea simplă și multiplă
4. Controlul accesului și moștenirea
5. Metode-constructor pentru clase derivate și cuvântul-cheie super
6. Supraîncărcarea (overloading) metodelor
7. Redefinirea (override) metodelor
8. Redefinirea parțială a unei metode

3. Moștenirea simplă și multiplă

Forma de mostenire folosita in Java este denumita **moștenire simplă** (**single inheritance**), deoarece fiecare clasa de obiecte **Java** nou creata poate fi derivata dintr-o singura superclasa (poate avea o singura superclasa).



3. Moștenirea simplă și multiplă

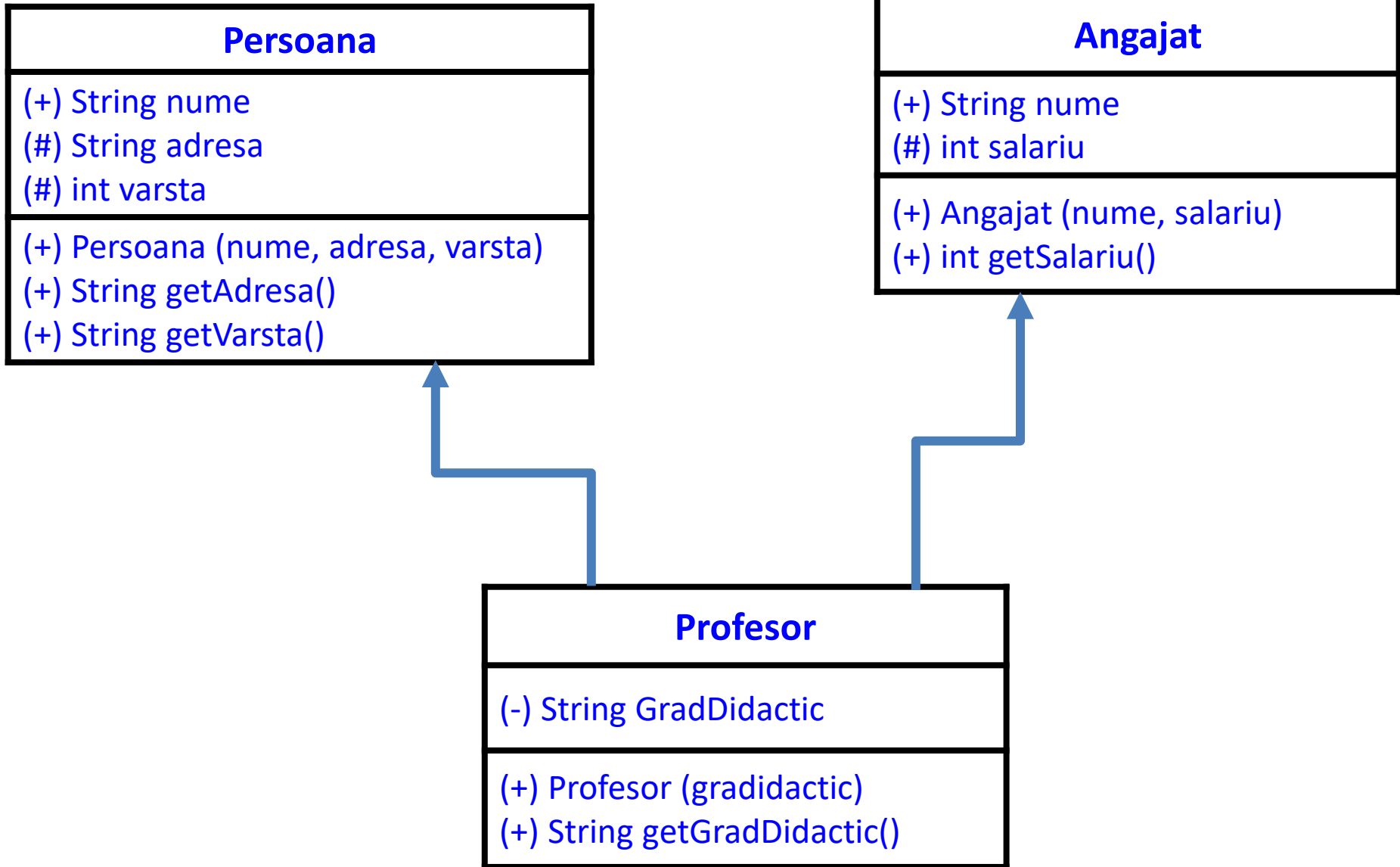
Moștenirea multiplă înseamnă ca *o clasă nou creată poate fi derivată din două sau mai multe superclase*, mostenind variabilele și metodele combinate din toate aceste superclase.

Mostenirea multiplă (**multiple inheritance**) oferă mijloace de creare a unor clase care cuprind aproape orice comportament imaginabil.

Acest lucru complică semnificativ definiția clasei și a codului necesar acesteia.

3. Moștenirea simplă și multiplă

Exemplu de **Moștenirea multiplă**:



3. Moștenirea simplă și multiplă

- **Moștenirea simplă poate fi restrictivă**, mai ales atunci când există un comportament similar, care trebuie duplicat pe diferite “ramuri” ale ierarhiei de clase (nu pe aceeași ramură a ierarhiei).
- Aceasta înseamnă că trebuie să dam definiții de metode despre ce înseamnă faptul că un obiect poate fi privit ca un mamifer sau ca un obiect spatio-temporal.

3. Moștenirea simplă și multiplă

- Dar, aceste definiții de metode sunt comune nu numai clasei de obiecte *Om* dar și altor clase de obiecte derivate sau nu din clasa *Om*, superclase sau nu ale clasei *Om*.
- Putem să găsim o multitudine de clase de obiecte ale căror instanțe pot fi privite ca obiecte spatio-temporale dar care să nu aibă mare lucru în comun cu omul (de exemplu clasa *Minge*).

3. Moștenirea simplă și multiplă

- O **clasa derivata** (numita si **subclasa**) mosteneste toate attributele (variabilele de instanta si de clasa) superclasei din care provine.
- **Clasa derivata** poate apoi:
 - sa adauge noi attribute
 - sa redefineasca metode ale superclasei
 - sau sa adauge noi metode
- Fiecare clasa derivata este o clasa complet noua.

Sintaxă folosită pentru a deriva o clasă nouă dintr-o superclasă

Pentru a declara o clasă derivată se folosește clauza *extends*, astfel:

```
[<modificatori_acces>] [<modificatori_clasa>] class  
<nume_subclasa> extends <nume_superclasa>  
{  
    <corpul_clasei>  
}
```


3. Moștenirea simplă și multiplă

unde:

- `<modificatori_acces>` - specifica domeniul de vizibilitate (folosire sau acces) al subclasei; modificatorul de acces poate fi: *public*;
- `<modificatori_clasa>` - specifica tipul subclasei definite; modificatorul clasei poate fi: *abstract, final*;
- `<nume_subclasa>` - specifica numele clasei derivate dintr-o superclasa;
- `<nume_superclasa>` - specifica numele unei superclase din care deriva subclasa;
- `<corpul_clasei>` - variabilele clasei și metodele care lucrează cu acestea, adăugate sau redefinite în subclasa.

3. Moștenirea simplă și multiplă

Observatii:

1. Orice metoda neprivata (adica publica, protejata (**protected**) sau prietenoasa (**friendly**)) din superclasa, care nu este redefinita (suprascrisa) in clasa derivata, este mostenita nemodificat, cu exceptia constructorilor. Metoda poate fi apoi apelata ca si cum ar face parte din clasa derivata.
2. Clasa derivata contine attribute si metode suplimentare (fata de cele mostenite din superclasa) care pot fi declarate: **private**, **protected**, **public** sau fara modificador de acces.

Mostenire

1. Conceptul de moștenire
2. Caracteristicile unei ierarhii de clase
3. Moștenirea simplă și multiplă
4. Controlul accesului și moștenirea
5. Metode-constructor pentru clase derivate și cuvântul-cheie super
6. Supraîncărcarea (overloading) metodelor
7. Redefinirea (override) metodelor
8. Redefinirea parțială a unei metode

4. Controlul accesului și moștenirea

- Ca regula generală, toți membrii de tip *public* ai superclasei devin membri de tip *public* ai clasei derivate.
- De asemenea, membrii de tip *private* ai superclasei sunt moșteniți, dar aceștia nu sunt accesibili în mod direct (adică, folosind operatorul “*punct*” sau *direct numele membrului*) în clasa derivată, ci prin intermediul altor metode publice moștenite de la superclasa care face posibil accesul.

4. Controlul accesului și moștenirea

Metodele speciale care examinează și modifică valoarea fiecărui atribut de tip *private* sunt denumite “**accesori**” și, respectiv, “**modificatori**”.

Se folosește convenția ca numele metodelor “**accesor**” să înceapă cu *get*, cum ar fi *getRaza()* din programul *TestCerc.java* prezentat în curs 12.

4. Controlul accesului și moștenirea

De asemenea, se folosește convenția ca numele metodelor “**modifier**” să înceapă cu **set**, cum ar fi *setRaza()* din programul *TestCerc.java* prezentat în curs 12.

Folosirea metodelor “**accesori**” și “**modifieri**” este foarte răspândită în programarea orientată obiect. Acest mod de abordare mărește gradul de reutilizare a codului, evitând folosirea lui necorespunzătoare.

Mostenire

1. Conceptul de moștenire
2. Caracteristicile unei ierarhii de clase
3. Moștenirea simplă și multiplă
4. Controlul accesului și moștenirea
5. Metode-constructor pentru clase derivate și cuvântul-cheie super
6. Supraîncărcarea (overloading) metodelor
7. Redefinirea (override) metodelor
8. Redefinirea parțială a unei metode

5. Metode-constructor pentru clase derivate și cuvântul-cheie *super*

Metodele-constructor nu se moștenesc.

Fiecare clasa derivata trebuie sa isi defineasca propriile metode-constructor.

Daca nu se defineste nici un constructor, **Java** va genera un constructor implicit (fara parametri).

5. Metode-constructor pentru clase derivate și cuvântul-cheie *super*

Acest constructor implicit al clasei derivate:

- va apela automat constructorul implicit (fara parametrii) al superclasei (aflata pe nivelul imediat superior) pentru membrii care au fost mosteniti, apoi
- va aplica initializarea implicita pentru attributele adaugate in clasa derivata (adica *0/false* pentru *tipurile primitive numerice/booleene* si *null* pentru *tipurile referinta*).

5. Metode-constructor pentru clase derivate și cuvântul-cheie *super*

- Asadar, *construirea unui obiect al unei clase derivate* are loc prin *construirea prealabila a portiunii mostenite* (constructorul clasei derivate apeleaza automat constructorul superclasei aflata pe nivelul imediat superior).
- Acest lucru este normal, deoarece mecanismul incapsularii afirma ca partea mostenita este un “intreg”, iar constructorul superclasei ne spune cum sa initializam acest “intreg”.

5. Metode-constructor pentru clase derivate și cuvântul-cheie *super*

Metodele-constructor ale superclasei pot fi apelate explicit in clasa derivata prin metoda **super()**.

*Metoda **super** poate sa apara doar in prima linie dintr-o metoda-constructor.*

Apelul unei metode-constructor a superclasei dintr-o clasa derivata, folosind **super** se face astfel:

super(<arg₁>, <arg₂>, <arg₃>, ...)

unde:

- **<arg₁>, <arg₂>, <arg₃>, ...** - specifica parametrii metodei-constructor a superclasei.

5. Metode-constructor pentru clase derivate și cuvântul-cheie *super*

De exemplu, sa presupunem ca o superclasa (ClasaSuper) are un constructor cu doi parametri de tip *int*.

Constructorul clasei derivate va avea, in general, forma:

```
public class ClasaDerivata extends ClasaSuper{  
    public ClasaDerivata(int x, int y){  
        super(x, y);  
        // alte instructiuni  
    }  
    ...  
}
```

Exemplu:

```
class Persoana
```

```
{
```

```
    private String nume;
```

```
    private String adresa;
```

```
    // constructor
```

```
    public Persoana (String nume, String adresa)
```

```
    {
```

```
        this.nume = nume;
```

```
        this.adresa = adresa;
```

```
    }
```

```
    public void afisare ()
```

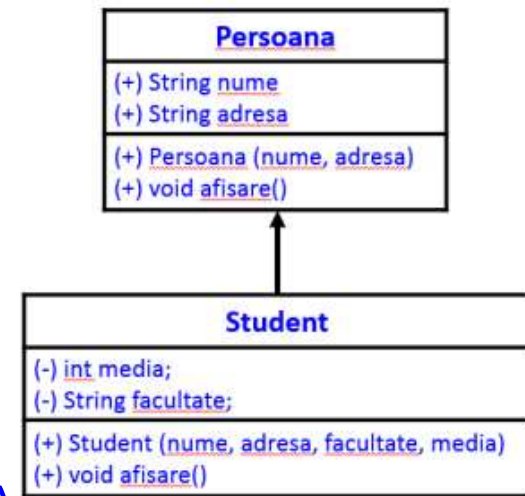
```
    {
```

```
        System.out.println ("Nume: " + nume);
```

```
        System.out.println ("Adresa: " + adresa);
```

```
    }
```

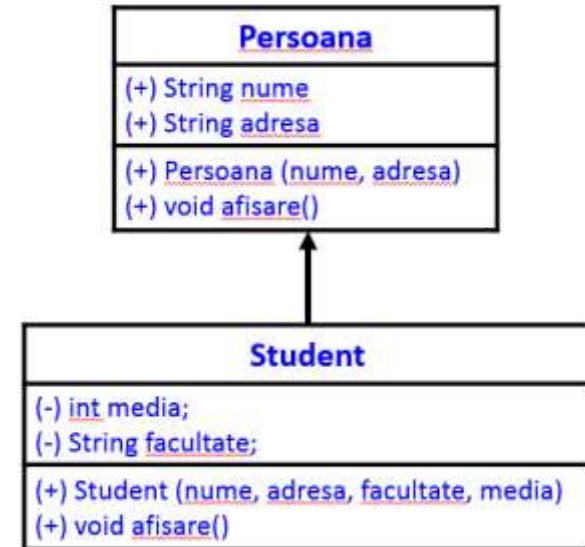
```
}
```



```

class Student extends Persoana
{
    private int media;
    private String facultate;
    public Student (String nume, String adresa, String
facultate, int media)
    {
        super (nume, adresa);
        this.media = media;
        this.facultate = facultate;
    }
    public void afisare ()
    {
        super.afisare ();
        System.out.println ("Facultate: " + facultate);
        System.out.println ("Media: " + media);
    }
}

```

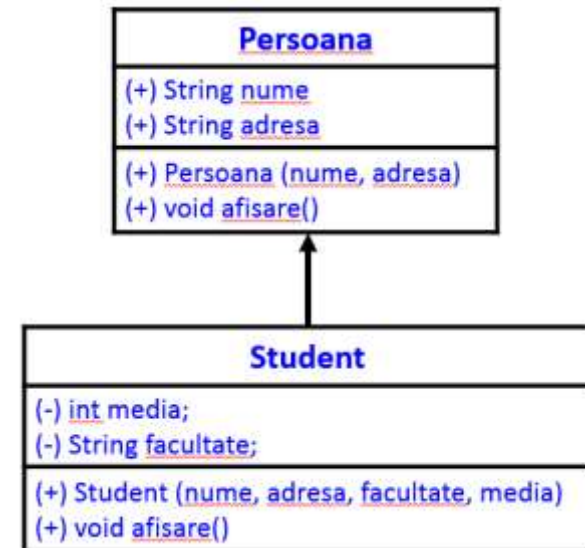


```
public class MyClass{
    public static void main(String args[]) {
        Student ob = new
Student("Popescu Ion", "Str. Victoriei nr. 13",
"Inginerie", 9);
        ob.afisare();
    }
}
```

Result

CPU Time: 0.23 sec(s), Memory: 33508 kilobyte(s)

```
Nume: Popescu Ion
Adresa: Str. Victoriei nr. 13
Facultate: Inginerie
Media: 9
```



API

Add compiler functionality to your application
Standards based REST API

[Goto API](#)

Online Compiler And Editor

76+ Languages with Multiple Versions and 2 DBs
Save, Share and Peer Programming
Embed to your Blog/Website

Search IDE/Compiler/Terminal

Java

Java (Advanced)

C

Online Java Compiler IDE

(Advanced IDE)

For simple single file programs and faster execution, use - [Basic Java IDE](#)

```
1 class Persoana
2 {
3     private String nume;
4     private String adresa;
5     // constructor
6     public Persoana (String nume, String adresa)
7     {
8         this.nume = nume;
9         this.adresa = adresa;
10    }
11    public void afisare ()
12    {
13        System.out.println ("Nume: " + nume);
14        System.out.println ("Adresa: " + adresa);
15    }
16 }
17 class Student extends Persoana
18 {
19     private int media;
20     private String facultate;
21     public Student (String nume, String adresa, String facultate, int media)
22     {
23         super (nume, adresa);
24         this.media = media;
25         this.facultate = facultate;
26     }
27     public void afisare ()
28     {
29         super.afisare ();
30         System.out.println ("Facultate: " + facultate);
31         System.out.println ("Media: " + media);
32     }
33 }
34
35 public class MyClass
36 {
37
38     public static void main(String args[])
39     {
40         Student ob = new Student("Popescu Ion", "Str. Victoriei nr. 13", "Inginerie", 9);
41         ob.afisare();
42     }
43 }
44 }
```


5. Metode-constructor pentru clase derivate și cuvântul-cheie *super*

Dupa cum se observa, daca trebuie apelat constructorul clasei de baza din clasa derivata, sintaxa este:

```
super (nume, adresa);
```

De asemenea, acest apel trebuie sa fie obligatoriu prima linie de cod din constructorul clasei derivate, altfel compilatorul va genera o eroare:

Call to super must be first statement in constructor

Mostenire

1. Conceptul de moștenire
2. Caracteristicile unei ierarhii de clase
3. Moștenirea simplă și multiplă
4. Controlul accesului și moștenirea
5. Metode-constructor pentru clase derivate și cuvântul-cheie super
6. **Supraîncărcarea (overloading) metodelor**
7. Redefinirea (override) metodelor
8. Redefinirea parțială a unei metode

6. Supraîncărcarea (**overloading**) metodelor

- **Supraîncărcarea metodelor** permite existența în interiorul aceleiași clase a mai multor *metode cu același nume, dar cu lista diferită de parametri* (ca număr și tip), deci *cu semnatura diferită*.
- **Supraîncărcarea metodelor** este permisă și dacă unele dintre metode sunt definite într-o superclasă și altele în clasele derivate din superclasă respectivă.

6. Supraîncărcarea (**overloading**) metodelor

- De exemplu, putem avea:
 - o metoda *int max(int a, int b)*
 - si o metoda *int max(int a, int b, int c)*,
 - ambele in cadrul aceleasi clase
 - sau putem defini prima metoda in cadrul unei superclase si cea de a doua metoda in cadrul unei clase derivate din superclasa.
- Atunci cand se face un apel al unei metode supraincarcate, compilatorul alege definitia de metoda examinand *lista parametrilor de apel* (adica, *semnatura*).

Mostenire

1. Conceptul de moștenire
2. Caracteristicile unei ierarhii de clase
3. Moștenirea simplă și multiplă
4. Controlul accesului și moștenirea
5. Metode-constructor pentru clase derivate și cuvântul-cheie super
6. Supraîncărcarea (overloading) metodelor
7. Redefinirea (override) metodelor
8. Redefinirea parțială a unei metode

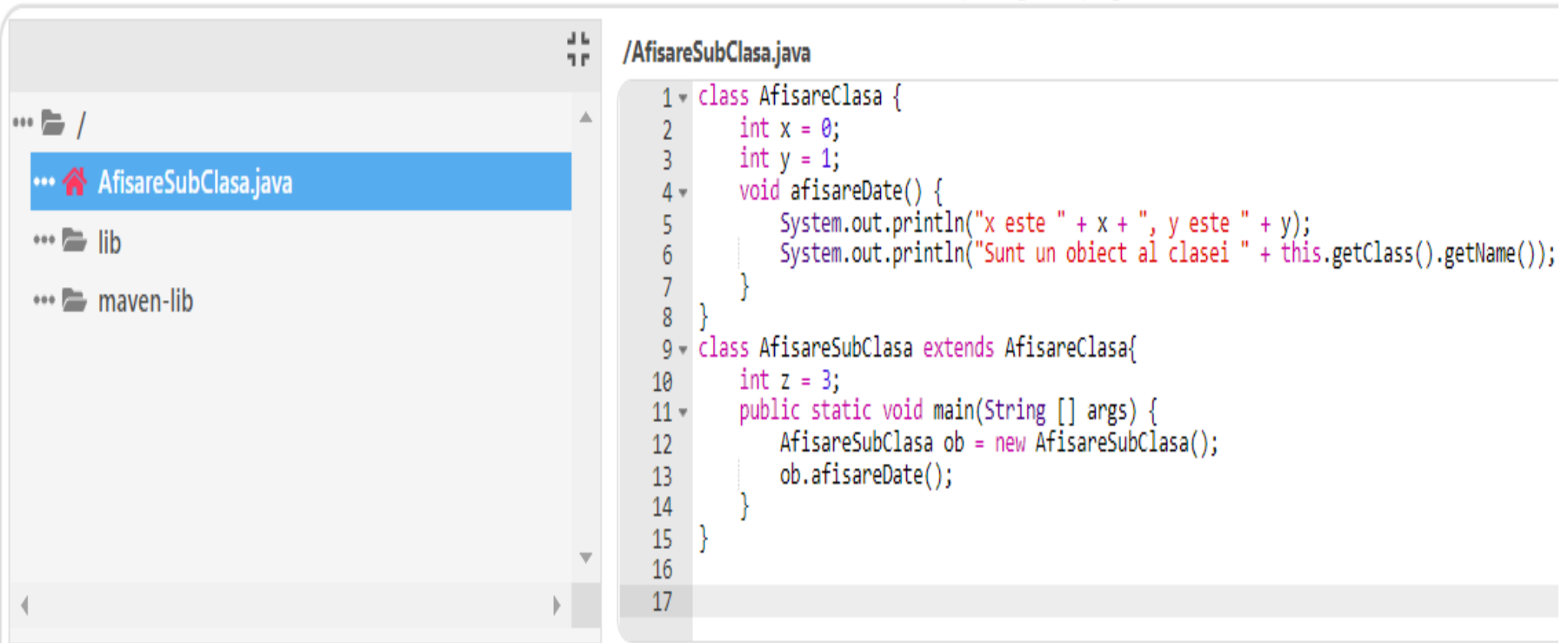
7. Redefinirea (**override**) metodelor

- **Redefinirea (suprascriere) metodelor** permite existenta in interiorul claselor derivate dintr-o superclasa a mai multor *metode cu acelasi nume, tip de rezultat si aceeasi lista de parametri*, deci *cu aceeasi semnatura*.
- Atunci cand se face un apel al unei metode redefinite, compilatorul foloseste definitia metodei care este gasita prima (incepand din partea de jos a ierarhiei de clase si mergand in sus).

Programul urmator ([AfisareSubClasa.java](#)) prezinta o clasa ce contine o metoda *afisareDate()*, care afiseaza numele clasei si valorile variabilelor de instanta.

- De asemenea, in acelasi fisier-sursa este inclusa si clasa denumita *AfisareSubClasa* derivata din clasa *AfisareClasa*.
- A fost creat un obiect de tip *AfisareSubClasa* si a fost apelata metoda *afisareDate()*.
- Deoarece clasa *AfisareSubClasa* nu defineste aceasta metoda, **Java** o cauta in superclasele clasei *AfisareSubClasa*, incepand cu superclasa *AfisareClasa*, unde gaseste metoda *afisareDate()* si o executa.

```
class AfisareClasa {  
    int x = 0;  
    int y = 1;  
    void afisareDate() {  
        System.out.println("x este " + x + ", y este " + y);  
        System.out.println("Sunt un obiect al clasei " +  
this.getClass().getName());  
    }  
}  
  
class AfisareSubClasa extends AfisareClasa {  
    int z = 3;  
    public static void main(String [] args) {  
        AfisareSubClasa ob = new AfisareSubClasa();  
        ob.afisareDate();  
    }  
}
```

```
/AfisareSubClasa.java
1 class AfisareClasa {
2     int x = 0;
3     int y = 1;
4     void afisareDate() {
5         System.out.println("x este " + x + ", y este " + y);
6         System.out.println("Sunt un obiect al clasei " + this.getClass().getName());
7     }
8 }
9 class AfisareSubClasa extends AfisareClasa{
10     int z = 3;
11     public static void main(String [] args) {
12         AfisareSubClasa ob = new AfisareSubClasa();
13         ob.afisareDate();
14     }
15 }
16
17
```

Result

CPU Time: 0.15 sec(s), Memory: 33768 kilobyte(s)

```
x este 0, y este 1
Sunt un obiect al clasei AfisareSubClasa
```

Deoarece metoda *afisareDate()* a superclasei *AfisareClasa* nu afiseaza si variabila de instanta z specifica subclasei *AfisareSubClasa*, metoda *afisareDate()* va fi redefinita (suprascrisa) in interiorul subclasei.

Iata noul program ([AfisareSubClasa.java](#)):

```
class AfisareClasa {  
    int x = 0;  
    int y = 1;  
    void afisareDate() {  
        System.out.println("x este " + x +  
", y este " + y);  
        System.out.println("Sunt un obiect  
al clasei " + this.getClass().getName());  
    }  
}
```

```
class AfisareSubClasa extends AfisareClasa {
    int z = 3;
    void afisareDate() {
        System.out.println("x este " + x + ", y
este " + y + ", z este " + z);
        System.out.println("Sunt un obiect al
clasei " + this.getClass().getName());
    }
    public static void main(String [] args) {
        AfisareSubClasa ob = new
AfisareSubClasa();
        ob.afisareDate();
    }
}
```

/AfisareSubClasa.java

```
1 class AfisareClasa {
2     int x = 0;
3     int y = 1;
4     void afisareDate() {
5         System.out.println("x este " + x + ", y este " + y);
6         System.out.println("Sunt un obiect al clasei " + this.getClass().getName());
7     }
8 }
9 class AfisareSubClasa extends AfisareClasa {
10     int z = 3;
11     void afisareDate() {
12         System.out.println("x este " + x + ", y este " + y + ", z este " + z);
13         System.out.println("Sunt un obiect al clasei " + this.getClass().getName());
14     }
15     public static void main(String [] args) {
16         AfisareSubClasa ob = new AfisareSubClasa();
17         ob.afisareDate();
18     }
19 }
```

... /
... AfisareSubClasa.java

... lib

... maven-lib

Rezultatul programului este:

Result

CPU Time: 0.21 sec(s), Memory: 35032 kilobyte(s)

```
x este 0, y este 1, z este 3  
Sunt un obiect al clasei AfisareSubClasa
```

Nota:

Apelul de metoda:

```
this.getClass().getName();
```

este folosit pentru aflarea numelui clasei din care face parte un obiect, in cazul de fata obiectul curent.

Mostenire

1. Conceptul de moștenire
2. Caracteristicile unei ierarhii de clase
3. Moștenirea simplă și multiplă
4. Controlul accesului și moștenirea
5. Metode-constructor pentru clase derivate și cuvântul-cheie super
6. Supraîncărcarea (overloading) metodelor
7. Redefinirea (override) metodelor
8. Redefinirea parțială a unei metode

8. Redefinirea parțială a unei metode

- **Redefinirea parțială** înseamnă ca *metoda din clasa derivată nu redefineste complet metoda cu aceeași semnătură din superclasa, ci extinde operațiile pe care aceasta le realizează.*
- Cu alte cuvinte, metoda din clasa derivată face ceva în plus față de cea originală din superclasa.
- Pentru a apela metoda din superclasa în interiorul metodei din clasa derivată se folosește referința ***super***.

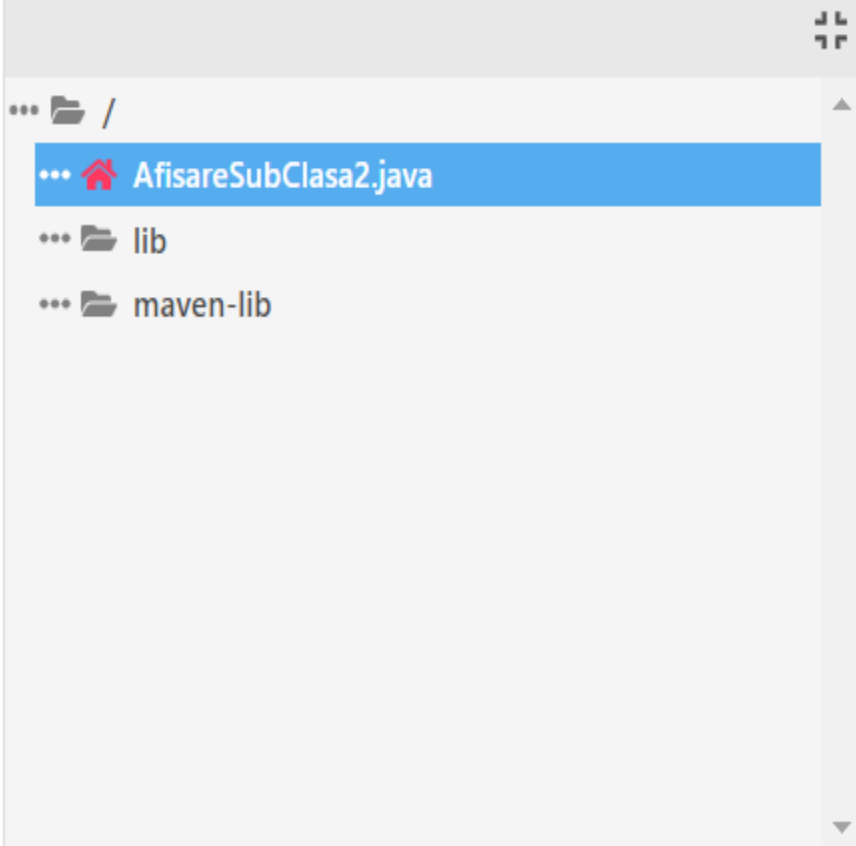
8. Redefinirea parțială a unei metode

Urmatorul program ([AfisareSubClasa2.java](#)) ilustreaza modul de redefinire partiala a metodei *afisareDate()* din subclasa *AfisareSubClasa1* si apelul in cadrul acesteia a metodei *afisareDate()* din superclasa *AfisareClasa1*.

```
class AfisareClasa1 {  
    int x = 0;  
    int y = 1;  
    void afisareDate() {  
        System.out.println("Sunt un obiect al clasei " +  
            this.getClass().getName());  
        System.out.println("x este " + x);  
        System.out.println("y este " + y);  
    }  
}
```

8. Redefinirea parțială a unei metode

```
class AfisareSubClasa2 extends AfisareClasa1 {
    int z = 3;
    void afisareDate() {
        super.afisareDate();
        System.out.println("z este " + z);
    }
    public static void main(String [] args) {
        AfisareSubClasa2 ob = new AfisareSubClasa2();
        ob.afisareDate();
    }
}
```



```
/AfisareSubClasa2.java
1 class AfisareClasa1 {
2     int x = 0;
3     int y = 1;
4     void afisareDate() {
5         System.out.println("Sunt un obiect al clasei " +
6             this.getClass().getName());
7         System.out.println("x este " + x);
8         System.out.println("y este " + y);
9     }
10 }
11
12 class AfisareSubClasa2 extends AfisareClasa1 {
13     int z = 3;
14     void afisareDate() {
15         super.afisareDate();
16         System.out.println("z este " + z);
17     }
18     public static void main(String [] args) {
19         AfisareSubClasa2 ob = new AfisareSubClasa2();
20         ob.afisareDate();
21     }
22 }
23
```

Result

CPU Time: 0.14 sec(s), Memory: 33624 kilobyte(s)

```
Sunt un obiect al clasei AfisareSubClasa2
x este 0
y este 1
z este 3
```

Accesul la metodele din superclasă și redefinirea metodelor în clasele derivate

Atunci când se creează o clasă derivată și se redefineste (suprascrie) o metodă declarată într-o superclasă, trebuie să se ia în considerare tipul de acces dat pentru metoda originală.

Astfel, in cazul metodelor mostenite, pentru care se doreste redefinirea (suprascrierea) in clasa derivata se impun urmatoarele reguli:

1. *metodele de tip **public*** dintr-o superclasa trebuie sa fie, de asemenea, de tip **public** in toate clasele derivate; ele nu pot fi redefinite de tip **private** in clasele derivate;
2. *metodele de tip **protected*** dintr-o superclasa pot fi de tip **protected** sau de tip **public** in clasele derivate; ele nu pot fi redefinite de tip **private** in clasele derivate;
3. *metodele de tip **private*** dintr-o superclasa nu pot fi redefinite (suprascrise) in clasele derivate.

Aplicația 1:

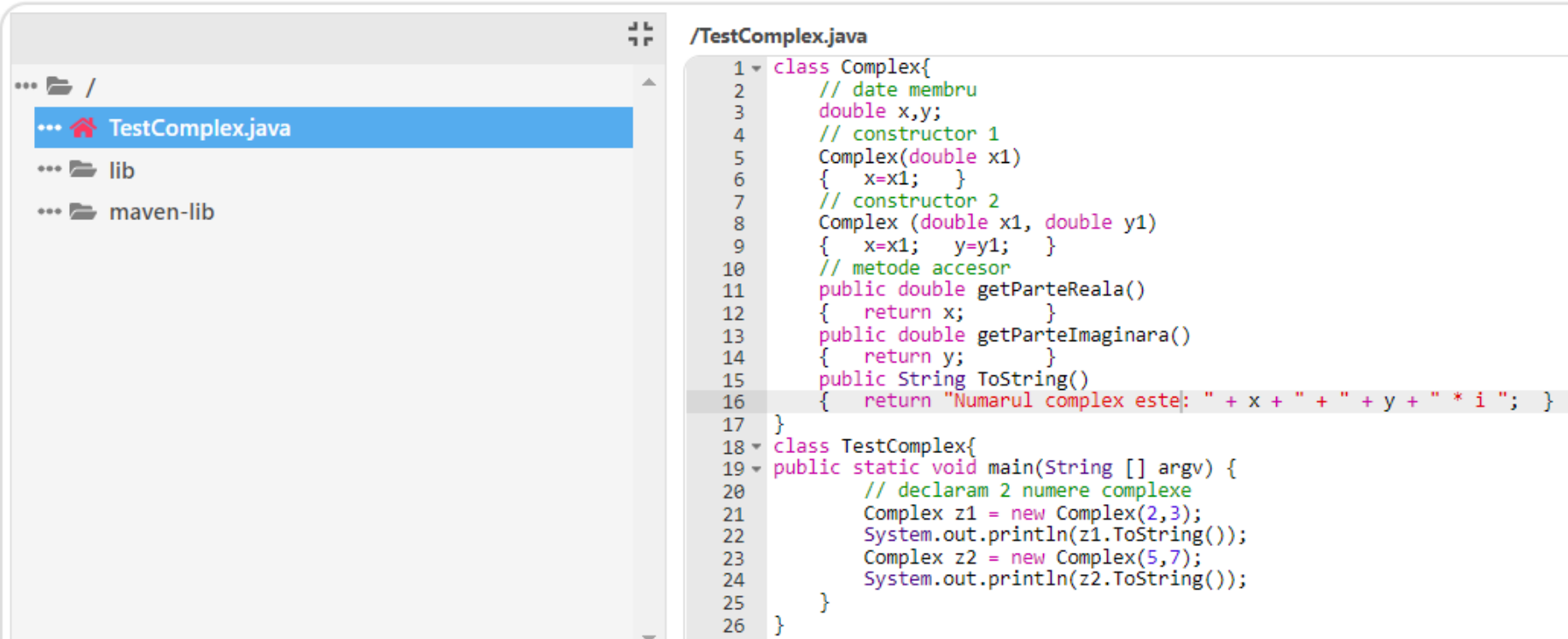
Se considera o clasa Complex (fișierul [Complex.java](#)) care gestionează operații cu numere complexe.

Clasa va avea:

- Doi constructori:
 - unul care să initializeze obiectele cu valori implicite
 - și altul care initializează variabilele cu valorile trimise ca parametri constructorului
- Două metode de tip accesoriu (pentru accesul la componentele clasei)
- O metodă de afișare

```
class Complex{
    // date membru
    double x,y;
    // constructor 1
    Complex(double x1)
    {
        x=x1;
    }
    // constructor 2
    Complex (double x1, double y1)
    {
        x=x1;  y=y1;
    }
    // metode accesori
    public double getParteReala()
    {
        return x;
    }
    public double getParteImaginara()
    {
        return y;
    }
    public String ToString()
    {
        return "Numarul complex este : " + x + "+" + y + " * i ";
    }
}
```

```
class TestComplex{  
    public static void main(String [] argv){  
        // declaram 2 numere complexe  
        Complex z1 = new Complex(2,3);  
        System.out.println(z1.ToString());  
        Complex z2 = new Complex(5,7);  
        System.out.println(z2.ToString());  
    }  
}
```

The screenshot shows an IDE window with a project explorer on the left and a code editor on the right. The project explorer shows a folder named 'TestComplex.java' selected, with subfolders 'lib' and 'maven-lib'. The code editor displays the following Java code:

```
/TestComplex.java
1 class Complex{
2     // date membru
3     double x,y;
4     // constructor 1
5     Complex(double x1)
6     { x=x1; }
7     // constructor 2
8     Complex (double x1, double y1)
9     { x=x1; y=y1; }
10    // metode accesori
11    public double getParteReala()
12    { return x; }
13    public double getParteImaginara()
14    { return y; }
15    public String ToString()
16    { return "Numarul complex este: " + x + " + " + y + " * i "; }
17 }
18 class TestComplex{
19 public static void main(String [] argv) {
20     // declaram 2 numere complexe
21     Complex z1 = new Complex(2,3);
22     System.out.println(z1.ToString());
23     Complex z2 = new Complex(5,7);
24     System.out.println(z2.ToString());
25 }
26 }
```

Result

CPU Time: 0.15 sec(s), Memory: 35612 kilobyte(s)

```
Numarul complex este: 2.0 + 3.0 * i
Numarul complex este: 5.0 + 7.0 * i
```

Aplicația 2:

Programul urmator ([TestPatrulater.java](#)) implementeaza clasa **Paralelogram** care mosteneste clasa **Patrulater** (fisierul [Patrulater.java](#)).

```
class Patrulater{  
    double xA,yA,xB,yB,xC,yC,xD,yD;  
    int valid;  
    public Patrulater(double x1,double  
y1,double x2,double y2,double x3,double  
y3,double x4,double y4)  
    {  
        xA=x1;        yA=y1;  
        xB=x2;        yB=y2;  
        xC=x3;        yC=y3;  
        xD=x4;        yD=y4;  
        valid_1();  
    }  
}
```

```
public int valid_1()
{
    if((xA!=xB) || (yA!=yB))&&((xA!=xC) || (yA!=yC))&&
((xA!=xD) || (yA!=yD))&&((xB!=xC) || (yB!=yC))&&((xB!=xD) ||
(yB!=yD)) && ((xC!=xD) || (yC!=yD)))
        valid=1;
    else valid=0;
    return valid;
}
void afis_1()
{
    if (valid==1) System.out.println("Figura este
patrulater.");
    else System.out.println("Figura NU este patrulater.");
}
};
```

Fisierul TestPatrulater.java

```
import java.io.*;
class Paralelogram extends Patrulater{
    double l1,l2,l3,l4,l5;
    int validp;
    public int valid_2()
    {
        valid_1();
        if (valid!=0) {
            l5=(xA-xB)*(xA-xB)+(yA-yB)*(yA-yB);
            l1=Math.sqrt(Math.abs(l5));
            l2=Math.sqrt(Math.abs((((xC-xB)*(xC-xB))+((yC-yB)*(yC-
yB))))));
            l3=Math.sqrt(Math.abs((((xC-xD)*(xC-xD))+((yC-
yD)*(yC-yD))))));
            l4=Math.sqrt(Math.abs((((xA-xD)*(xA-xD))+((yA-
yD)*(yA-yD))))));
            if ((l1==l3)&&(l2==l4)) validp=1;
                else validp=0;
        }
        else validp=0;
        return validp;
    }
}
```

```
public void afis_2()  
    {  
        if (validp!=0) System.out.println("Figura este  
paralelogram.");  
        else System.out.println("Figura NU este  
paralelogram.");;  
    }  
    public Paralelogram(double x1, double y1, double x2,  
double y2, double x3, double y3, double x4, double y4)  
        {  
            super(x1,y1,x2,y2,x3,y3,x4,y4);  
            valid_2();  
        }  
};
```

```
class TestPatrulater{
    public static void main(String [] argv) throws IOException {
        Paralelogram pp = new Paralelogram(0.0,0.0,1.0,0.0,1.0,1.0,0.0,1.0);
        pp.valid_1();
        pp.afis_1();
        pp.valid_2();
        pp.afis_2();
        System.out.println("Dati coordonatele varfurilor unei alte figuri
geometrice:");
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(isr);
        String s1 = br.readLine();
        pp.xA = Double.parseDouble(s1);
        System.out.println("xA = "+pp.xA);
        String s2 = br.readLine();
        pp.yA = Double.parseDouble(s2);
        System.out.println("yA = "+pp.yA);
        String s3 = br.readLine();
        pp.xB = Double.parseDouble(s3);
        System.out.println("xB = "+pp.xB);
        String s4 = br.readLine();
        pp.yB = Double.parseDouble(s4);
        System.out.println("yB = "+pp.yB);
    }
}
```

(continuare)

```
String s5 = br.readLine();  
pp.xC = Double.parseDouble(s5);  
System.out.println("xC = "+pp.xC);  
String s6 = br.readLine();  
pp.yC = Double.parseDouble(s6);  
System.out.println("yC = "+pp.yC);  
String s7 = br.readLine();  
pp.xD = Double.parseDouble(s7);  
System.out.println("xD = "+pp.xD);  
String s8 = br.readLine();  
pp.yD = Double.parseDouble(s8);  
System.out.println("yD = "+pp.yD);  
pp.valid_1();  
pp.afis_1();  
pp.valid_2();  
pp.afis_2();
```

```
}
```

```
}
```


/TestPatrulater.java

```
1 import java.io.*;
2 class Patrulater{
3     double xA,yA,xB,yB,xC,yC,xD,yD;
4     int valid;
5     public Patrulater(double x1,double y1,double x2,double y2,double x3,double y3,double x4,double y4)
6     {
7         xA=x1; yA=y1;
8         xB=x2; yB=y2;
9         xC=x3; yC=y3;
10        xD=x4; yD=y4;
11        valid_1();
12    }
13    public int valid_1()
14    {
15        if(((xA!=xB)|| (yA!=yB))&&((xA!=xC)|| (yA!=yC))&& ((xA!=xD)|| (yA!=yD))&&((xB!=xC)|| (yB!=yC))&&((xB!=xD)|| (yB!=yD)) && ((xC!=xD)|| (yC!=yD)))
16            valid=1;
17        else valid=0;
18        return valid;
19    }
20    void afis_1()
21    {
22        if (valid==1) System.out.println("Figura este patrulater.");
23        else System.out.println("Figura NU este patrulater.");
24    }
25 }
26
27 class Paralelogram extends Patrulater{
28     double l1,l2,l3,l4,l5;
29     int validp;
30     public int valid_2()
31     {
32         valid_1();
33         if (valid!=0) {
34             l5=(xA-xB)*(xA-xB)+(yA-yB)*(yA-yB);
35             l1=Math.sqrt(Math.abs(l5));
36             l2=Math.sqrt(Math.abs((((xC-xB)*(xC-xB))+((yC-yB)*(yC-yB)))));
37             l3=Math.sqrt(Math.abs((((xC-xD)*(xC-xD))+((yC-yD)*(yC-yD)))));
38             l4=Math.sqrt(Math.abs((((xA-xD)*(xA-xD))+((yA-yD)*(yA-yD)))));
39             if ((l1==l3)&&(l2==l4)) validp=1;
40             else validp=0;
41         }
42         else validp=0;
43         return validp;
44     }
45 }
```

```

45
46     public void afis_2()
47     {
48         if (validp!=0) System.out.println("Figura este paralelogram.");
49         else System.out.println("Figura NU este paralelogram.");
50     }
51     public Paralelogram(double x1, double y1, double x2, double y2, double x3, double y3, double x4, double y4)
52     {
53         super(x1,y1,x2,y2,x3,y3,x4,y4);
54         valid_2();
55     }
56 }
57
58 class TestPatrulater{
59     public static void main(String [] argv) throws IOException
60     {
61         Paralelogram pp = new Paralelogram(0.0,0.0,1.0,0.0,1.0,1.0,0.0,1.0);
62         pp.valid_1();
63         pp.afis_1();
64         pp.valid_2();
65         pp.afis_2();
66
67         System.out.println("Dati coordonatele varfurilor unei alte figuri geometrice:");
68         InputStreamReader isr = new InputStreamReader(System.in);
69         BufferedReader br = new BufferedReader(isr);
70
71         String s1 = br.readLine(); pp.xA = Double.parseDouble(s1); System.out.println("xA = "+pp.xA);
72         String s2 = br.readLine(); pp.yA = Double.parseDouble(s2); System.out.println("yA = "+pp.yA);
73         String s3 = br.readLine(); pp.xB = Double.parseDouble(s3); System.out.println("xB = "+pp.xB);
74         String s4 = br.readLine(); pp.yB = Double.parseDouble(s4); System.out.println("yB = "+pp.yB);
75         String s5 = br.readLine(); pp.xC = Double.parseDouble(s5); System.out.println("xC = "+pp.xC);
76         String s6 = br.readLine(); pp.yC = Double.parseDouble(s6); System.out.println("yC = "+pp.yC);
77         String s7 = br.readLine(); pp.xD = Double.parseDouble(s7); System.out.println("xD = "+pp.xD);
78         String s8 = br.readLine(); pp.yD = Double.parseDouble(s8); System.out.println("yD = "+pp.yD);
79
80         pp.valid_1();
81         pp.afis_1();
82         pp.valid_2();
83         pp.afis_2();
84     }
85 }
86

```

Execute Mode, Version, Inputs & Arguments

JDK 17.0.1

Interactive

Stdin Inputs

10
10
10
20
0
0
0
10

CommandLine Arguments

 Execute



Result

CPU Time: 0.15 sec(s), Memory: 34036 kilobyte(s)

```
Figura este patrulater.  
Figura este paralelogram.  
Dati coordonatele varfurilor unei alte figuri geometrice:  
xA = 10.0  
xA = 10.0  
xB = 10.0  
yB = 20.0  
xC = 0.0  
yC = 0.0  
xD = 0.0  
yD = 10.0  
Figura este patrulater.  
Figura NU este paralelogram.
```

Întrebări?