

## Moștenirea

### 1. Scopul lucrării

Obiectivele de învățare ale acestei sesiuni de laborator sunt:

- Înțelegerea conceptului de constructor al superclasei și folosirea lui corespunzătoare
- Înțelegerea conceptelor de supraîncărcare (overloading) și suprascriere (overriding) a unei metode
- Înțelegerea problemelor care apar în legătură cu moștenirea și a tehnicilor de evitare

### 2 `super(...)` – constructorul superclasei (părintelui)

Un obiect are câmpurile proprii clase plus câmpurile clasei părinte, ale clasei bunic ș.a.m.d. până sus la la clasa rădăcină, clasa `Object`.

Este necesar să *inițializați toate câmpurile*, de aceea trebuie *invocați toți constructorii*!

Compilerul Java inserează automat apelurile de constructori necesare în procesul de înlănțuire al constructorilor sau o puteți face explicit.

Compilerul Java inserează un apel la constructorul părintelui (**super**) dacă nu există un apel de constructor ca primă instrucțiune al constructorului Dvs. Fiind dat codul care urmează

```
public class Point
{
    int m_x;
    int m_y;

    //===== Constructor
    public Point(int x, int y)
    {
        m_x = x;
        m_y = y;
    }

    //===== Constructor implicit fara parametri
    public Point()
    {
        this(0, 0); // Apeleaza celalalt constructor.
    }
    . . .
}
```

Cele ce urmează sunt echivalente cu constructorul de mai sus.

```
//===== Constructor (ca in apelul de mai sus)
public Point(int x, int y) {
    super(); // realizat automat daca nu apelati constructorul.
    m_x = x;
    m_y = y;
}
```

#### 2.1 De ce am dori să apelăm explicit constructorul `super`?

Există două situații în care este necesar apelul explicit al constructorului superclasei:

1. Doriți să apelați constructorul cu parametri (apelul generat automat nu are parametri).
2. Nu există un constructor fără parametri; sunt definiți doar constructori cu parametri. În clasa părinte.

Fiecare obiect conține variabile instanță ale clasei sale. Ceea ce nu este la fel de evident este că fiecare obiect are și toate variabilele instanță ale tuturor superclaselor sale (ale tuturor strămoșilor). Aceste variabile ale superclaselor trebuie inițializate înainte de a inițializa variabilele instanță ale claselor.

### 2.1.1 Inserarea automată a apelului constructorului superclasei

La crearea unui obiect este nevoie să apelăm toți constructorii tuturor superclaselor ca să-și inițializeze câmpurile proprii. Java face acest lucru automat la început dacă nu-l faceți dvs.

Spre exemplu, primul constructor `Point` ar fi putut fi scris

```
public Point(int xx, int yy)
{
    super(); // inserat automat
    x = xx;
    y = yy;
}
```

### 2.1.2 Apeluri explicite la constructorul superclasei

În mod normal nu se scrie explicit apelul constructorului superclasei dar există două situații când este necesar:

**Transmiterea parametrilor.** Doriți să invocați un constructor al clasei părinte care are parametri (constructorul implicit nu are parametri). Spre exemplu, dacă definiți o subclasă a lui `JFrame` poate faceți astfel:

```
class MyWindow extends JFrame {
    . . .
    //===== constructor
    public MyWindow(String title) {
        super(title);
    }
    . . .
}
```

În exemplul de mai sus ați dorit să folosiți constructorul lui `JFrame` care primește ca parametru titlul cadrului. Ar fi fost simplu să lăsați să fie apelat constructorul implicit și să folosiți o metodă care setează titlul, ca alternativă.

```
class MyWindow extends JFrame {
    . . .
    //===== constructor
    public MyWindow(String title) {
        // constructorul implicit al superclasei este automat inserat
        setTitle(title); // apelează metoda din superclasa.
    }
    . . .
}
```

**Nu există constructor fără parametri.** Nu există în părinte un constructor fără parametri. Uneori nu are sens să creăm un obiect fără a furniza parametri. Spre exemplu, ar trebuie într-adevăr să existe un constructor `Point` fără parametri? Deși exemplul anterior a definit un constructor fără parametri pentru a ilustra folosirea lui `this`, nu este probabil o idee bună pentru `Point`.

**Exemplu de clasă care nu are constructor fără parametri**

```

////////// clasa fara constructor fara parametri.
// Daca este definit un constructor, compilatorul nu creaza automat un
// constructor fara parametri.
class Parent
{
    int _x;
    Parent(int x) { // constructor
        _x = x;
    }
}

////////// clasa care trebuie sa apeleze super in constructor
class Child extends Parent
{
    int _y;
    Child(int y) { // GRESIT (eroare de compilare), este nevoie de apel explicit la super.
        _y = y;
    }
}

```

În exemplul de mai sus nu există un apel explicit la un constructor în prima linie a constructorului, așa că compilatorul va insera un apel la un constructor fără parametri al părintelui, dar așa ceva nu există! De aceea, acest fapt duce la o eroare de compilare. Problema poate fi rezolvată prin schimbarea clasei **Child**.

```

////////// clasa care trebuie sa invoce super in constructor
class Child extends Parent
{
    int _y;
    Child(int y) { // CORECT, se face apel explicit la super cu parametri.
        super(0);
        _y = y;
    }
}

```

Sau, clasa **Parent** poate defini un constructor fără parametri.

```

////////// clasa fara constructor fara parametri.
// Daca este definit un constructor, compilatorul nu creaza automat un
// constructor fara parametri.
class Parent
{
    int _x;
    Parent(int x)
    { // constructor cu un parametru
        _x = x;
    }

    Parent()
    { // constructor fara parametri
        _x = 0;
    }
}

```

O cale mai bună de a defini un constructor fără parametri este prin apelul unui constructor parametrizat astfel că orice schimbări sunt făcute într-un singur constructor.

```

    Parent()
    { // constructor fara parametri
        this(0);
    }

```

Remarcați că fiecare dintre acești constructori apelează implicit constructorul fără parametri al părintelui clasei sale ș. a. m. d. până ce se ajunge în final la clasa **Object**.

### 2.1.3 Cum se suprascrie o metodă?

Pentru a suprascrie o metodă într-o clasă nouă, pur și simplu se reproduce numele, lista de argumente și tipul returnat al metodei originale într-o definiție nouă de metodă într-o nouă clasă. Apoi se scrie corpul metodei. Codul din corp se scrie astfel încât să facă comportamentul metodei suprascrise să fie corespunzător pentru un obiect din clasa nouă.

Iată o descriere mai precisă a suprascrierii metodelor luată din cartea *The Complete Java 2 Certification Study Guide*, de Roberts, Heller și Ernest:

*"O suprascriere validă are ordinea și tipul argumentelor identică, tip returnat identic și nu este mai puțin accesibilă decât metoda originală. Metoda care suprascrie nu trebuie să arunce nici o excepție verificată care nu a fost declarată pentru metoda originală."*

Orice metodă nedeclarată **final** poate fi suprascrisă într-o subclasă.

### 2.1.4 Suprascriere versus supraîncărcare

Nu confundați suprascrierea cu supraîncărcarea metodelor. Iată ce spun autorii menționați despre supraîncărcarea metodelor:

*"O supraîncărcare validă diferă ca număr sau tip de argumente. Diferențele de nume de argumente nu contează. Este permis un tip returnat diferit, dar acesta singur nu este suficient pentru a distinge o metodă care supraîncarcă."*

## 2.2 Probleme uzuale legate de moștenire

### 2.2.1 Ascunderea variabilelor

Atunci când atât clasa părinte cât și subclasa au un câmp cu același nume, avem de a face cu acoperirea/umbrirea (shadowing) variabilei. Dacă câmpul din clasa părinte are acces **private** sau se află în alt pachet și are acces implicit, nu e loc de confuzie. Clasa fiică nu poate accesa câmpul respectiv din clasa părinte. Așa că e clar cu ce variabilă se vor face operațiile în clasa fiică.

Totuși, dacă o variabilă cu același nume din clasa părinte este accesibilă instanțelor clasei fiice, există o serie de reguli ne-intuitive care determină care câmp este accesat de diferitele invocări. Regula generală este că variabila accesată depinde de clasa la care s-a asignat tipul variabilei.

Spre exemplu, Listing A conține două clase, **Base** și **Sub**, care reprezintă o relație clasă părinte/clasă copil. Ambele clase au un câmp întreg numit **field**. Listing B ilustrează instanțierea clasei **Sub** urmată de mai multe modalități legale de referire la una sau alta dintre variabilele **field**.

Listing A

```
public class Base
{
    public int field = 0;
    public int getField() { return field; }
}
public class Sub extends Base
{
    public int field = 1;
    public int getField() { return field; }
}
```

---

Listing B

---

```
Sub s = new Sub();
Base b = s;
System.out.println(s.field); // access one
System.out.println(b.field); // access two
System.out.println(((Sub)b).field); // access three
System.out.println(((Base)s).field); // access four
```

---

Prima dintre aceste tehnici, marcată `access one`, accesează direct variabila `field` din instanța lui `Sub` numită `s`. Acest acces, cum ne așteptam, dă valoarea 1. În schimb, `access two`, ilustrează deconectarea logică implicată de ascunderea variabilei. Chiar dacă expresia `b==s` este adevărată, cel de-al doilea acces, `b.field`, se evaluează la 0.

Această diferență între `b.field` și `s.field`, în ciuda echivalenței de referire dintre `b` și `s`, arată clar câmpul miniat inerent ascunderii variabilelor. Singura diferență dintre `b` și `s` în acest exemplu este tipul de variabilă în care sunt stocate.

Expresiile marcate `access three` și `access four` urmează regula de tip care dictează valoarea. Specific, `access three` dă valoarea 1 deoarece obiectul `b` este convertit la tipul `Sub` înainte de verificarea variabilei sale `field`. Asemănător, `access four` dă valoarea 0 datorită faptului că obiectul `s` este convertit la tipul `Base` înainte de accesul lui `field`.

## 2.2.2 Suprascrierea metodelor

În timp ce câmpurile numite la fel din subclase ascund cele de același nume în clasele părinte, se folosește o terminologie diferită pentru a trata probleme similare referitoare la metode. Atunci când o clasă părinte și o clasă copil au fiecare o metodă cu aceeași semnătură, metoda din clasa copil suprascrie metoda din clasa părinte. În Listing A, se poate vedea că metoda `getField` din clasa `Sub` are același nume și parametri – adică nici un parametru – ca metoda `getField` din clasa `Base`.

---

Listing C

---

```
Sub s = new Sub();
Base b = s;
System.out.println(s.getField()); // access one
System.out.println(b.getField()); // access two
System.out.println(((Sub)b).getField()); // access three
System.out.println(((Base)s).getField()); // access four
```

---

Listing C ilustrează aceeași instanțiere de obiect ca Listing B și multe invocări posibile ale celor două metode `getField`. Linia marcată `access one` din Listing C invocă metoda `getField` pe o referință la singurul obiect din exemplu stocată într-o variabilă de tipul `Sub`. Cum a fost cazul din exemplul asemănător din Listing B, această invocare dă valoarea 1.

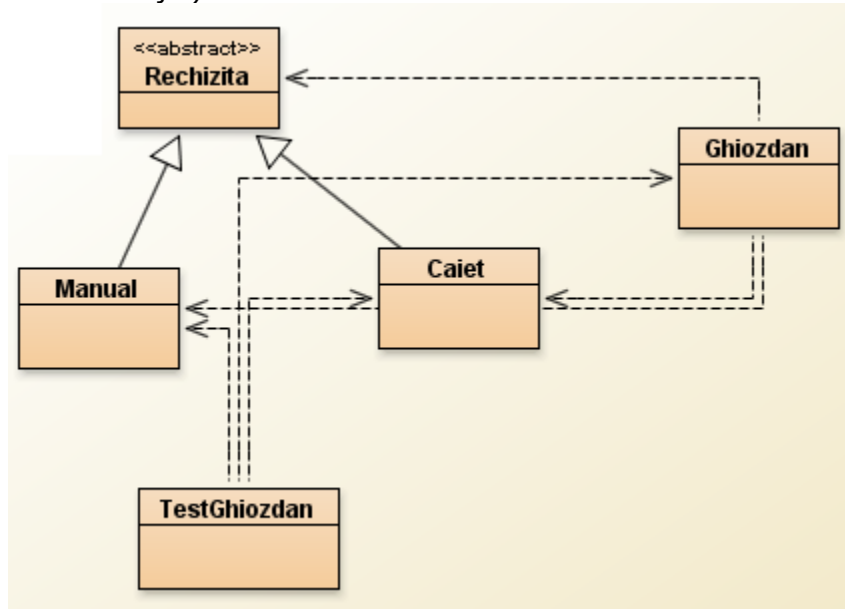
Diferența dintre suprascrierea metodei și ascunderea variabilei se observă la compararea lui `access two` din Listing C cu cel numit la fel din Listing B. Pe când în Listing B s-a accesat câmpul `field` al clasei părinte, în Listing C este accesată variabila `field` din subclasă, în ciuda tipului de variabilă în care este stocată referința; rezultă valoarea 1.

Această aderare strictă la identitatea referință/instanță față de tipul variabilei este sufletul *polimorfismului*. Polimorfismul este caracteristica care delegează comportamentul la clasa reală a instanței referite — nu la tipul în care este stocată referința. Polimorfismul este legat numai de metode.

Continuând cu exemplele, ajungem la `access three` și `access four` din Listing C. Amândouă produc valoarea 1. Iarăși, aceasta este din cauza naturii *polimorfe* a metodelor. Pentru a accesa variabila ascunsă din clasa `Base` prin invocarea unei metode, trebuie folosită sintaxa `super.field` în clasa `Sub`.

### 3 Mersul lucrării

- 3.1 Studiați textul și sursele prezentate în lucrare.
- 3.2 Implementați un program care să simuleze lucrul cu un ghiozdan cu rechizite (vezi diagrama de clase de mai jos).



Soluția creată trebuie să conțină 5 clase:

- O clasă abstractă **Rechizita** cu următoarele proprietăți:
  - o Un atribut **Etichetă** – variabila care va descrie numele rechizitei.
  - o O metodă abstractă **getNumE()**
- Două subclase **Manual** și **Caiet** ce extind clasa abstractă rechizita. Ambele clase vor implementa metoda **getNumE()** într-un mod diferit. Spre exemplu, ambele vor returna valoarea etichetei din Rechizita, însă pentru a face diferența de implementare pentru fiecare clasă în parte se va adăuga substringul "Caiet" sau "Manual" în numele etichetei.
- O clasă **Ghiozdan** care va conține:
  - o O listă de rechizite (implementată cu tipul de date tablou sau ArrayList).
  - o metoda **add** pentru adăugarea unui **Caiet** în lista de rechizite
  - o metoda **add** pentru adăugarea unui **Manual** în lista de rechizite
  - o metoda **listItems** pentru listarea tuturor rechizitelor
  - o metoda **listManual** pentru listarea doar a manualelor din listă
  - o metoda **listCaiet** pentru listarea doar a caietelor din listă
  - o metoda **getNrRechizite** pentru afisarea numărului de rechizite
  - o metoda **getNrManuale** pentru a calcularea și afisarea nr. de manuale din listă
  - o metoda **getNrCaiete** pentru a calcularea și afisarea nr. de Caiete din listă

- O clasă **TestGhiozdan** cu metoda main în care se vor efectua următoarele operații:
  - o Instanțierea unui obiect de tip **Ghiozdan**
  - o Crearea mai multor obiecte de tip **Caiet** și **Manual**, și adăugarea lor în lista de rechizite.
  - o Afișarea nr. total de rechizite, nr. de caiete și nr. de manuale.
  - o Listarea tuturor rechizitelor.
  - o Listarea doar a rechizitelor de tip **Caiet**.
  - o Listarea doar a rechizitelor de tip **Manual**.

3.3. Implementați o versiune simplificată a jocului de sah care să verifice corectitudinea mutărilor pe tabla de sah a diferitelor piese, și să efectueze mutarea atunci când mutarea este validă.

Indicații de implementare:

- Creați o clasă abstractă **PiesaSah** care să fie caracterizată prin atributele *numePiesa* și coordonatele ei pe tabla de sah: *coordX* și *coordY*, și care să conțină metoda abstractă *mutaPiesa(int new\_coordX, int new\_coordY)*.
- Creați apoi câte o clasă pentru fiecare piesă de sah (**Nebun**, **Tura**, **Pion** etc.) care să mostenească clasa **PiesaSah** și să implementeze metoda *mutaPiesa* în funcție de regulile de deplasare a fiecărei piese.
- Într-o clasă de test, creați tabla de sah ca un tablou de piese de sah, poziționați câteva piese pe tabla de sah și apoi încercați să efectuați operații de mutare a pieselor la coordonate noi citite din consolă. Piesa se va muta la noile coordonate doar dacă mutarea este validă.

3.4. Studiați și executați codul care urmează. Remarcați ordinea de inițializare.

```
class Insecta
{
    private int i = 9;
    protected int j;
    Insecta()
    {
        System.out.println("i = " + i + ", j = " + j);
        j = 39;
    }

    private static int x1 = printInit("static Insecta.x1 initializata");

    static int printInit(String s)
    {
        System.out.println(s);
        return 47;
    }
}

public class Gindac extends Insecta
{
    private int k = printInit("Gindac.k initializat");
    public Gindac()
    {
        System.out.println("k = " + k);
        System.out.println("j = " + j);
    }

    private static int x2 = printInit("static Gindac.x2 initializat");

    public static void main(String[] args)
    {
        System.out.println("Gindac constructor");
        Gindac b = new Gindac();
    }
}
```

```
/* Output:  
static Insect.x1 initialized  
static Gindac.x2 initialized  
Gindac constructor  
i = 9, j = 0  
Gindac.k initialized  
k = 47  
j = 39  
*/
```

- 3.5. Adăugați un alt tip de insectă și apoi subclase ale lui Gindac și a tipului Dvs. Adăugați cod pentru a arăta ordinea operațiilor de inițializare pentru tipurile Dvs.