# Moştenirea

## 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. Constructorul superclasei (părintelui) – super(...)

Un obiect are câmpurile propriei clase plus câmpurile clasei părinte, ale clasei bunic ş.a.m.d. până sus la la clasa rădăcină, clasa Object.
Este necesar să *iniţializaţi toate câmpurile*, de aceea trebuie *invocaţi toţi constructorii*!
Compilatorul Java inserează automat apelurile de constructori necesare în procesul de înlănţuire al constructorilor sau o puteţi face explicit.
Compilatorul Java inserează un apel la constructorul părintelui (**super**) dacă nu există un apel de constructor ca primă instrucţiune al constructorului Dvs. Fiind dat codul care urmează

```
public class Point
{
    int x;
    int y;

    //============ Constructor
    public Point(int xx, int yy)
    {
        x = xx;
        y = yy;
    }

    //============ 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 xx, int yy) {
    super();  // realizat automat daca nu apelati explicit super-constructorul
    x = xx;
    y = yy;
}
```

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

Există două situaţii în care este necesar apelul explicit al constructorului superclasei:
1. Doriţi să apelaţi constructorul cu parametri (apelul generat automat nu are parametri).
2. Nu există un constructor fără parametri; sunt definiţi doar constructori cu parametri în clasa părinte.

Fiecare obiect conţine variabile instanţă ale clasei sale. Ceea ce nu este la fel de evident este că fiecare obiect are şi toate variabilele instanţă ale tuturor superclaselor sale (ale tuturor strămoşilor). Aceste variabile ale superclaselor trebuie iniţializate înainte de a iniţializa variabilele instanţă ale claselor.

### 2.1.1 Inserarea automată a apelului constructorului superclasei

La crearea unui obiect este nevoie să apelăm toţi constructorii tuturor superclaselor ca să-şi iniţializeze câmpurile proprii. Java face acest lucru automat la început dacă nu-l faceţi 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 situaţii când este necesar:
**Transmiterea parametrilor**. Doriţi să invocaţi un constructor al clasei părinte care are parametri (constructorul implicit nu are parametri). Spre exemplu, dacă definiţi o subclasă a lui **`JFrame`** poate faceţi astfel:

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

În exemplul de mai sus aţi dorit să folosiţi constructorul lui **`JFrame`** care primeşte ca parametru titlul cadrului. Ar fi fost simplu să lăsaţi să fie apelat constructorul implicit şi să folosiţi 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, trebuie într-adevăr să existe un constructor **`Point`** fără parametri? Deşi 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 creeaza 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, aşa că compilatorul va insera un apel la un constructor fără parametri al părintelui, dar aşa 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 cu 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);
    }
```

Remarcaţi că fiecare dintre aceşti constructori apelează implicit constructorul fără parametri al părintelui clasei sale ş.a.m.d. până ce se ajunge în final la clasa **`Object`**.

## 2.2 Suprascrierea și supraîncărcarea metodelor

### 2.2.1 Cum se suprascrie o metodă?

Pentru a suprascrie o metodă într-o clasă nouă, pur şi simplu se reproduce numele, lista de argumente şi tipul returnat al metodei originale într-o definiţie 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 şi Ernest:

> *"O suprascriere validă are ordinea şi tipul argumentelor identică, tip returnat identic şi nu este mai puţin 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.

### 2.2.2 Suprascriere versus supraîncărcare

Nu confundaţi suprascrierea cu supraîncărcarea metodelor. Iată ce spun autorii menţionaţi anterior despre supraîncărcarea metodelor:

> *"O supraîncărcare validă diferă ca număr sau tip de argumente. Diferenţele 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."*

## 2.3 Probleme uzuale legate de moştenire

### 2.3.1 Ascunderea variabilelor

Atunci când atât clasa părinte cât şi subclasa au un câmp cu acelaşi nume, avem de a face cu acoperirea/umbrirea (*shadowing*) variabilei. Dacă câmpul din clasa părinte are acces **`private`** sau se află în alt pachet şi are acces implicit, nu e loc de confuzie. Clasa fiică nu poate accesa câmpul respectiv din clasa părinte. Aşa că e clar cu ce variabilă se vor face operaţiile în clasa fiică.
Totuşi, dacă o variabila cu acelaşi nume din clasa părinte este accesibila instanţelor 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 conţine două clase, **`Base`** şi **`Sub`**, care reprezintă o relaţie clasă părinte/clasă copil. Ambele clase au un câmp întreg numit **`field`**. Listing B ilustrează instanţierea clasei **`Sub`** urmată de mai multe modalităţi 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 instanţa lui **Sub** numită **s**. Acest acces, cum ne aşteptam, 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 diferenţă între **`b.field`** şi **`s.field`**, în ciuda echivalenţei de referire dintre **b** şi **s**, arată clar câmpul minat inerent ascunderii variabilelor. Singura diferenţă dintre **b** şi **s** în acest exemplu este tipul de variabilă în care sunt stocate.
Expresiile marcate `access three` şi `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`**.

### 2.3.2 Suprascrierea metodelor

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

Listing C

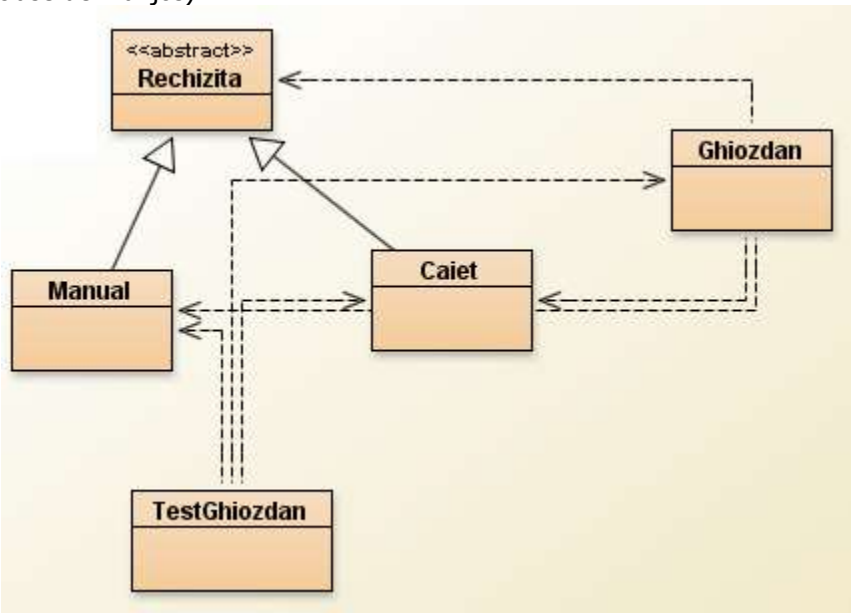
```
Sub s = new Sub();
Base b = s;
System.out.println(s.getField()); // access one
System.out.println(b.getField()); // access two
System.out.println(((Sub)b).getField()); // access three
System.out.println(((Base)s).getField()); // access four
```

Listing C ilustrează aceeaşi instanţiere de obiect ca Listing B şi multe invocări posibile ale celor două metode **`getField`**. Linia marcată `access one` din Listing C invocă metoda **`getField`** pe o referinţă 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.
Diferenţa dintre suprascrierea metodei şi 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 referinţa; rezultă valoarea 1.
Această aderare strictă la identitatea referinţă/instanţă faţă de tipul variabilei este sufletul *polimorfismului*. Polimorfismul este caracteristica care deleagă comportamentul la clasa reală a instanţei referite — nu la tipul în care este stocată referinţa. Polimorfismul este legat numai de metode.
Continuând cu exemplele, ajungem la `access three` şi `access four` din Listing C. Amândouă produc valoarea 1. Iarăşi, aceasta este din cauza naturii *polimorfice* 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 Studiaţi textul şi sursele prezentate în lucrare.

3.2 Implementaţi un program care să simuleze lucrul cu un ghiozdan cu rechizite (vezi diagrama de clase de mai jos).




Soluţia creată trebuie să conţină 5 clase:

- O clasă abstractă **Rechizita** care va conține:
  - un atribut **etichetă** – variabilă care va descrie numele rechizitei
  - o metodă abstractă **getNume()**
- Două subclase **Manual** şi **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 diferenţa de implementare pentru fiecare clasă în parte se va adăuga *substring*-ul "Caiet" sau "Manual" în numele etichetei
- O clasă **Ghiozdan** care va conţine:
  - o listă de rechizite (implementată cu tipul de date tablou)
  - metoda **add** pentru adăugarea unui **Caiet** în lista de rechizite
  - metoda **add** pentru adăugarea unui **Manual** în lista de rechizite
  - metoda **listItems** pentru listarea tuturor rechizitelor
  - metoda **listManual** pentru listarea doar a manualelor din listă
  - metoda **listCaiet** pentru listarea doar a caietelor din listă
  - metoda **getNrRechizite** pentru afisarea numărului de rechizite
  - metoda **getNrManuale** pentru a calcularea şi afisarea nr. de manuale din listă
  - metoda **getNrCaiete** pentru a calcularea şi afisarea nr. de caiete din listă

- O clasă **TestGhiozdan** cu metoda main în care se vor efectua următoarele operaţii:
  - instanţierea unui obiect de tip **Ghiozdan**
  - crearea mai multor obiecte de tip **Caiet** şi **Manual**, şi adăugarea lor în lista de rechizite
  - afişarea nr. total de rechizite, nr. de caiete şi nr. de manuale
  - listarea tuturor rechizitelor
  - listarea doar a rechizitelor de tip **Caiet**
  - listarea doar a rechizitelor de tip **Manual**

3.3. Implementaţi o versiune simplificată a jocului de sah care să verifice corectitudinea mutărilor pe tabla de șah a diferitelor piese, și să efectueze mutarea atunci când mutarea este validă.
Indicații de implementare:
- creați o clasa abstractă **PiesaSah** care să fie caracterizată prin atributele *numePiesa* și coordonatele ei pe tabla de sah: *coordX* si *coordY*, și care să conțină metoda abstractă *mutaPiesa(int new_coordX, int new_coordY)*
- creați apoi câte o clasa pentru fiecare piesă de șah (**Nebun**, **Tura**, **Pion** etc.) care să moștenească clasa PiesaSah și să implementeze metoda mutaPiesa în funcție de regulile de deplasare a fiecărei piese
- într-o clasa de test, creați tabla de șah ca un tablou de piese de șah, 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. Studiaţi şi executaţi codul care urmează. Remarcaţi ordinea de iniţializare.

```
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 Insecta.x1 initializata
static Gindac.x2 initializat
Gindac constructor
i = 9, j = 0
Gindac.k initializat
k = 47
j = 39
*/
```

3.5. Adăugaţi un alt tip de insectă şi apoi subclase ale lui Gindac şi a tipului Dvs. Adăugaţi cod pentru a arăta ordinea operaţiilor de iniţializare pentru tipurile Dvs.