Agregare și moștenire

- Responsabili: Laurențiu Stamate
- Data publicării: 07.10.2017
- Data ultimei modificări: 07.10.2017

Obiective

Scopul acestui laborator este familiarizarea studenților cu noțiunile de **agregare** și de **moștenire** a claselor.

Aspectele urmărite sunt:

- studierea mecanismului de moștenire
- înțelegerea diferenței între moștenire și agregare
- downcasting și upcasting

Agregare și Compunere

Agregarea și compunerea se referă la prezența unei referințe pentru un obiect într-o altă clasă. Acea clasă practic va refolosi codul din clasa corespunzatoare obiectului. Exemplu:

Compunere:

```
public class Foo {
    private Bar bar = new Bar();
}
```

Agregare:

```
public class Foo {
    private Bar bar;

    // The Bar object can continue to exist even if the Foo object doesn't
exist
    Foo(Bar bar) {
       this.bar = bar;
    }
}
```

Exemplu practic:

```
class Page {
    private String content;
    public int numberOfPages;
```

```
    public Page(String content, int numberOfPages) {
       this.content        = content;
       this.numberOfPages  = numberOfPages;
    }
}

class Book {
    private String title;           // Composition
    private Page[] pages;           // Composition
    private LibraryRow libraryRow = null;   // Aggregation

    public Book(int size, String title, LibraryRow libraryRow) {
        this.libraryRow = libraryRow;
        this.title = title;

        pages = new Page[size];

        for (int i = ; i < size; i++) {
           pages[i] = new Page("Page " + i, i);
        }
    }
}

class LibraryRow {
    private String rowName = null;        // Aggregation

    public LibraryRow(String rowName) {
        this.rowName = rowName;
    }
}

class Library {

    public static void main(String[] args) {
        LibraryRow row = new LibraryRow("a1");
        Book book = new Book(100, "title", row);

        // After there is no reference to the Book object, the
        // Garbage Collector will delete (at a certain time, not
        // necessarily immediately) that instance, but the LibraryRow
        // object passed to the constructor is not affected.

        book = null;
    }
}
```

- **Agregarea** (aggregation) - obiectul-container poate exista și în absența obiectelor agregate de aceea este considerată o *asociere slabă* (*weak association*). În exemplul de mai sus, un raft de bibliotecă poate exista și fără cărți.
- **Compunerea** (composition) - este o agregare *puternică* (*strong*), indicând că existența unui obiect este dependentă de un alt obiect. La dispariția obiectelor conținute prin compunere,

existența obiectului container încetează. În exemplul de mai sus, o carte nu poate exista fără pagini.

Inițializarea obiectelor conținute poate fi făcută în 3 momente de timp distincte:

- la **definirea** obiectului (înaintea constructorului: folosind fie o valoare inițială, fie blocuri de inițializare)
- în cadrul **constructorului**
- chiar **înainte de folosire** (acest mecanism se numește inițializare leneșă (*lazy initialization*))

Moștenire (Inheritance)

Numită și **derivare**, moștenirea este un mecanism de refolosire a codului specific limbajelor orientate obiect și reprezintă posibilitatea de a defini o clasă care **extinde** o altă clasă deja existentă. Ideea de bază este de a **prelua** funcționalitatea existentă într-o clasă și de a **adăuga** una nouă sau de a o **modela** pe cea existentă.

Clasa existentă este numită **clasa-părinte**, **clasa de bază** sau **super-clasă**. Clasa care extinde clasa-părinte se numește **clasa-copil (child)**, **clasa derivată** sau **sub-clasă**.

Spre deosebire de C++, Java nu permite *moștenire multiplă* (*multiple inheritance*), astfel că nu putem întâlni ambiguități de genul Problema Rombului / Diamond Problem. Mereu când vom vrea să ne referim la metoda părinte (folosind cuvântul cheie `super`, cum vom vedea mai jos), acel părinte este unic determinat.

Agregare vs. moștenire

Când se folosește moștenirea și când agregarea?

Răspunsul la această întrebare depinde, în principal, de datele problemei analizate dar și de concepția designerului, neexistând o rețetă general valabilă în acest sens.
În general, **agregarea** este folosită atunci când se dorește folosirea trăsăturilor unei clase în interiorul altei clase, dar nu și interfața sa (prin moștenire, noua clasă ar expune și metodele clasei de bază). Putem distinge două cazuri:

- uneori se dorește implementarea funcționalității obiectului conținut în noua clasă și **limitarea** acțiunilor utilizatorului doar la metodele din noua clasă (mai exact, se dorește să nu se permită utilizatorului folosirea metodelor din vechea clasă). Pentru a obține acest efect se va **agrega** în noua clasă un obiect de tipul clasei conținute și având specificatorul de acces `private`.
- obiectul conținut (agregat) trebuie/se dorește a fi accesat **direct**. În acest caz vom folosi specificatorul de acces `public`. Un exemplu în acest sens ar fi o clasă numită `Car` care conține ca membrii publici obiecte de tip `Engine`, `Wheel` etc.

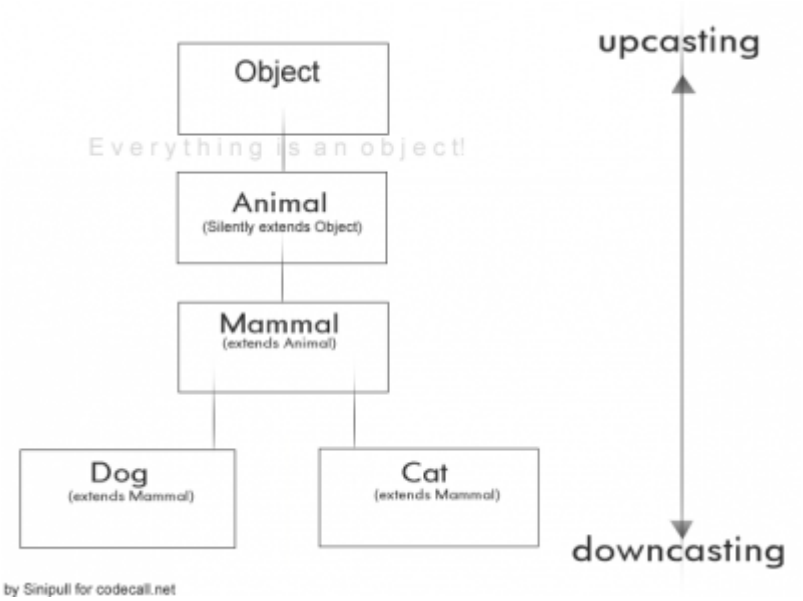
Moștenirea este un mecanism care permite crearea unor versiuni “specializate” ale unor clase existente (de bază). Moștenirea este folosită în general atunci când se dorește construirea unui tip de date care să reprezinte o implementare specifică (o specializare oferită prin clasa derivată) a unui lucru mai general. Un exemplu simplu ar fi clasa `Dacia` care moștenește clasa `Car`.

Diferența dintre moștenire și agregare este de fapt diferența dintre cele 2 tipuri de relații majore prezente între obiectele unei aplicații :

- **is a** - indică faptul că o clasă este derivată dintr-o clasă de bază (intuitiv, dacă avem o clasă `Animal` și o clasă `Dog`, atunci ar fi normal să avem `Dog` derivat din `Animal`, cu alte cuvinte `Dog` *is an* `Animal`)
- **has a** - indică faptul că o clasă-container are o clasă conținută în ea (intuitiv, dacă avem o clasă `Car` și o clasă `Engine`, atunci ar fi normal să avem `Engine` referit în cadrul `Car`, cu alte cuvinte `Car` *has a* `Engine`)

Upcasting și Downcasting

Convertirea unei referințe la o clasă derivată într-una a unei clase de bază poartă numele de **upcasting**. Upcasting-ul este facut **automat** și **nu** trebuie declarat explicit de către programator.



Exemplu de upcasting:

```
class Instrument {
    public void play() {}

    static void tune(Instrument i) {
        i.play();
    }
}

// Wind objects are instruments
// because they have the same interface:
public class Wind extends Instrument {
    public static void main(String[] args) {
        Wind flute = new Wind();
        Instrument.tune(flute); // !! Upcasting automatically because the
method
                                // gets an Instrument object, not a Wind
```

```
object
                                // So it would be redundant to make an
explicit cast like
                                // Instrument.tune((Instrument) flute)
    }
}
```

Deși obiectul `flute` este o instanță a clasei `Wind`, acesta este pasat ca parametru în locul unui obiect de tip `Instrument`, care este o superclasa a clasei `Wind`. Upcasting-ul se face la pasarea parametrului. Termenul de **upcasting** provine din diagramele de clase (în special UML) în care moștenirea se reprezintă prin 2 blocuri așezate unul sub altul, reprezentând cele 2 clase (sus este clasa de bază iar jos clasa derivată), unite printr-o săgeată orientată spre clasa de bază.

Downcasting este operația **inversă** upcast-ului și este o conversie explicită de tip în care se merge în **jos** pe ierarhia claselor (se convertește o clasă de bază într-una derivată). Acest cast trebuie făcut **explicit** de către programator. Downcasting-ul este **posibil** numai dacă obiectul declarat ca fiind de o clasă de bază este, de fapt, instanță clasei derivate către care se face downcasting-ul.

Iată un exemplu în care este folosit downcasting:

```
class Animal {
    public void eat() {
        System.out.println("Animal eating");
    }
}

class Wolf extends Animal {
    public void howl() {
        System.out.println("Wolf howling");
    }

    public void eat() {
        System.out.println("Wolf eating");
    }
}

class Snake extends Animal {
    public void bite() {
        System.out.println("Snake biting");
    }
}

class Test {
    public static void main(String[] args) {
        Animal a [] = new Animal[2];

        a[] = new Wolf();    // Upcasting automatically
        a[1] = new Snake();   // Upcasting automatically

        for (int i = ; i < a.length; i++) {
            a[i].eat(); // 1
```

```
            if (a[i] instanceof Wolf) {
                ((Wolf)a[i]).howl(); // 2
            }

            if (a[i] instanceof Snake) {
                ((Snake)a[i]).bite(); // 3
            }
        }
    }
}
```

Codul va afișa:

```
Wolf eating
Wolf howling
Animal eating
Snake biting
```

În liniile marcate cu **2** și **3** se execută un downcast de la `Animal` la `Wolf`, respectiv `Snake` pentru a putea fi apelate metodele specifice definite în aceste clase. Înaintea execuției downcast-ului (conversia de tip la Wolf respectiv Snake) verificăm dacă obiectul respectiv este de tipul dorit (utilizând operatorul **`instanceof`**). Dacă am încerca să facem downcast către tipul `Wolf` al unui obiect instantiat la `Snake` mașina virtuală ar semnala acest lucru aruncând o excepție la rularea programului.

Apelarea metodei `eat()` (linia **1**) se face direct, fără downcast, deoarece această metodă este definită și în clasa de bază `Animal`. Datorită faptului că `Wolf` suprascrie (*overrides*) metoda `eat()`, apelul `a[0].eat()` va afișa "Wolf eating". Apelul `a[1].eat()` va apela metoda din clasă de bază (la ieșire va fi afișat "Animal eating") deoarece `a[1]` este instantiat la `Snake`, iar `Snake` nu suprascrie metoda `eat()`.

Upcasting-ul este un element foarte important. De multe ori răspunsul la întrebarea: *este nevoie de moștenire?* este dat de răspunsul la întrebarea: *am nevoie de upcasting?* Aceasta deoarece upcasting-ul se face atunci când pentru unul sau mai multe obiecte din clase derivate se execută aceeași metodă definită în clasa părinte.

Să încercăm să evităm folosirea instanceof

Totuși, deși v-am ilustrstat cum `instanceof` ne poate ajuta să ne dăm seama la ce să facem **downcasting**, este de preferat să ne organizăm clasele și designul codului în așa fel încât să lăsăm limbajul Java să facă automat verificarea tipului și să cheme metoda corespunzătoare. Vom refactororiza codul anterior pentru a nu fi nevoie de `instanceof`:

```
class Animal {
    public void eat() {
        System.out.println("Animal eating");
    }
```

```
    public void action() {
        // we need this method because we will create a vector
        // of Animal instances and we will call this method on them
    }
}

class Wolf extends Animal {
    public void action() {    // we call it howl in action
        System.out.println("Wolf howling");
    }

    public void eat() {
        System.out.println("Wolf eating");
    }
}

class Snake extends Animal {
    public void action() {    // we call it bite in action
        System.out.println("Snake biting");
    }
}

class Test {
    public static void main(String[] args) {
        Animal a [] = new Animal[2];

        a[] = new Wolf();
        a[1] = new Snake();

        for (int i = ; i < a.length; i++) {
            a[i].eat();

            // now that they are called the same, we can call the action
method
            // from the Animal class (notice why it was a need to define the
            // action method in the Animal class), and the appropriate
method
            // will be called for the actual type of a[i] instance

            a[i].action();
        }
    }
}
```

Codul va afișa:

```
Wolf eating
Wolf howling
Animal eating
Snake biting
```

Implicații ale moștenirii

În Java, clasele și membrii acestora (metode, variabile, clase interne) pot avea diverși specificatori de acces, prezentați pe wiki în Organizarea surselor și controlul accesului.

- specificatorul de acces **`protected`** - specifică faptul că membrul sau metoda respectivă poate fi accesată doar din cadrul clasei înseși sau din clasele derivate din această clasă. Clasele nu pot avea acest specificator, doar membrii acestora!
- specificatorul de acces **`private`** - specifică faptul că membrul sau metoda respectivă poate fi accesată doar din cadrul **clasei** înseși, nu și din clasele derivate din această clasă. Clasele nu pot avea acest specificator, doar membrii acestora!

Constructorii **nu** se moștenesc și pot fi apelați doar în contextul unui constructor copil. Apelurile de constructor sunt înlănțuite, ceea ce înseamnă că înainte de a se inițializa obiectul copil, mai întâi se va inițializa obiectul părinte. În cazul în care părintele este copil la rândul lui, se va înițializa părintele lui (până se va ajunge la parintele suprem – root).

În laboratorul Constructori și referințe au fost prezentate și câteva din cuvintele cheie ce pot fi puse înaintea unor membri ai claselor, sau chiar a claselor.

- cuvântul cheie **`final`**
  - folosit la declararea unei metode, implicând faptul că metoda nu poate fi suprascrisă în clasele derivate
  - folosit la declararea unei clase, implicând faptul că acea clasă nu poate fi derivată (de exemplu clasa String)

Pe lângă reutilizarea codului, moștenirea dă posibilitatea de a dezvolta pas cu pas o aplicație (procedeul poartă numele de *incremental development*). Astfel, putem folosi un cod deja funcțional și adaugă alt cod nou la acesta, în felul acesta izolându-se bug-urile în codul nou adăugat. Pentru mai multe informații citiți capitolul *Reusing Classes* din cartea *Thinking în Java (Bruce Eckel)*

Suprascrierea, supraîncărcarea si ascunderea metodelor statice

Suprascrierea (*overriding*) presupune înlocuirea funcționalității din clasa/clasele părinte pentru instanța curentă.
Supraîncărcarea (*overloading*) presupune furnizarea de funcționalitate în plus, fie pentru metodele din clasa curentă, fie pentru clasa/clasele părinte.

```
public class Car {
    public void print() {
        System.out.println("Car");
    }

    public void init() {
        System.out.println("Car");
    }
```

```
    public void addGasoline() {
        // do something
    }
}

class Dacia extends Car {
    public void print() {
        System.out.println("Dacia");
    }

    public void init() {
        System.out.println("Dacia");
    }

    public void addGasoline(Integer gallons) {
        // do something
    }

    public void addGasoline(Double gallons) {
        // do something
    }
}
```

Metodele dependente de instanță sunt poliformice (la runtime pot avea diferite implementări) deci ele pot fi suprascrise sau supraîncarcăte. Metoda `print` este **suprascrisă** în clasa `Dacia` ceea ce înseamnă că orice instanță, chiar dacă se face cast la tipul `Car` metoda ce se va apela va fi mereu metoda `print` din clasa `Dacia`. Metoda `addGasoline` este **supraîncărcată** ceea ce înseamnă că putem executa metode cu semnături diferite dar același nume (cel mai folosit in crearea metodelor de conversie).

```
Car a = new Car();
Car b = new Dacia();
Dacia c = new Dacia();
Car d = null;

a.print(); // prints Car
b.print(); // prints Dacia
c.print(); // prints Dacia
d.print(); // throws NullPointerException
```

Suprascrierea nu se aplică și metodelor statice pentru că ele nu sunt dependente de instanță. Dacă în exemplul de mai sus facem metodele print din Car și din Dacia statice, rezultatul va fi următorul:

```
Car a = new Car();
Car b = new Dacia();
Dacia c = new Dacia();
Car d = null;

a.print(); // prints Car
b.print(); // prints Car because the declared type of b is Car
c.print(); // prints Dacia because the declared type of c is Dacia
```

```
d.print(): // prints Car because the declared type of b is Car
```

O să punem accent pe aceste concepte în laboratorul visitor

Sintaxa Java permite apelarea metodelor statice pe instanțe (e.g. a.print în loc de Car.print), dar acest lucru este considerat bad practice pentru că poate îngreuna înțelegerea codului.

Cuvântul cheie super. Întrebuințări

Cuvântul cheie `super` se referă la instanța părinte a clasei curente. Acesta poate fi folosit în două moduri: apelând o metoda suprascrisă (*overriden*) sau apelând constructorul părinte.

Apelând o metodă suprascrisă

```
public class Superclass {

    public void printMethod() {
        System.out.println("Printed in Superclass.");
    }
}

public class Subclass extends Superclass {

    // overrides printMethod in Superclass
    public void printMethod() {
        super.printMethod();  // calls the parent method

        System.out.println("Printed in Subclass.");
    }

    public static void main(String[] args) {
        Subclass s = new Subclass();
        s.printMethod();
    }
}
```

Codul va afișa:

```
Printed in Superclass.
Printed in Subclass.
```

Apelând constructorul părinte

```
class Superclass {
```

```
    public Superclass() {
       System.out.println("Printed in Superclass constructor with no
args.");
    }

    public Superclass(int a) {
       System.out.println("Printed in Superclass constructor with one
integer argument.");
    }
}

class Subclass extends Superclass {
    public Subclass() {
        super();    // calls the parent constructor
                    // this call must be on the first line !!

        System.out.println("Printed in Subclass constructor with no args.");
    }

    public Subclass(int a) {
        super(a);   // calls the parent constructor
                    // this call must be on the first line !!

        System.out.println("Printed in Subclass constructor with one integer
argument.");
    }

    public static void main(String[] args) {
        Subclass s1 = new Subclass(20);
        Subclass s2 = new Subclass();
    }
}
```

Codul va afișa:

```
Printed in Superclass constructor with one integer argument.
Printed in Subclass constructor with one integer argument.
Printed in Superclass constructor with no args.
Printed in Subclass constructor with no args.
```

Invocarea constructorului părinte **trebuie** să fie prima linie dintr-un constructor al unei subclase, dacă invocarea părintelui există (se poate foarte bine să nu apelăm `super` din constructor).

Chiar dacă nu se specifică apelul metodei `super()`, compilatorul va apela automat constructor-ul implicit al părintelui însă dacă se dorește apelarea altui constructor, apelul de `super(args)` respectiv este obligatoriu

Exerciții

Exercițiile 1-5 se rezolvă în ordine

1. (**2p**) Întrucât în ierarhia de clase Java, clasa `Object` se află în rădăcina arborelui de moștenire pentru orice clasă, orice clasă va avea acces la o serie de facilități oferite de `Object`. Una dintre ele este metoda `toString()`, al cărei scop este de a oferi o reprezentare a unei instanțe de clasă sub forma unui șir de caractere.
   - Definiți clasa **`Form`** cu un membru `color` de tip `String`, o metoda `getArea()` care pentru început va intoarce 0 și o metodă `toString()` care va returna acestă culoare.
   - Clasa va avea, de asemenea:
     - un constructor fără parametri
     - un constructor ce va inițializa culoarea.
   - Din ea derivați clasele **`Triangle`** și **`Circle`**:
     - Clasa `Triangle` va avea 2 membri `height` si `base` de tip `float`.
     - Clasa `Circle` va avea membrul `radius` de tip `float`.
     - Clasele vor avea:
       - constructori fără parametri
       - constructori care permit inițializarea membrilor. Identificați o modalitate de **reutilizare** a codului existent.
   - Instanțiati clasele `Triangle` și `Circle`, și apelați metoda `toString()` pentru fiecare instanță.
   - suprascrieti metoda `getArea()` pentru a intoarce aria specifica figuri geometrice.
2. (**2p**) Adăugați metode `toString()` în cele două clase derivate, care să returneze tipul obiectului, culoarea si aria. De exemplu:
   - pentru clasa `Triangle`, se va afișa: “Triunghi: rosu 10”
   - pentru clasa `Circle`, se va afișa: “Cerc: verde 12.56”
   - Modificați implementarea `toString()` din clasele derivate astfel încât aceasta să utilizeze implementarea metodei `toString()` din clasa de bază.
3. (**1p**) Adăugați o metodă `equals` în clasa `Triangle`. Justificați criteriul de echivalentă ales.
   - Hint: vedeți metodele clasei Object, moștenită de toate clasele - Object are metoda `equals`, a cărei implementare verifică echivalența obiectelor comparând referințele.
4. (**1p**) Upcasting.
   - Creați un vector de obiecte `Form` și populați-l cu obiecte de tip `Triangle` și `Circle` (upcasting).
   - Parcurgeți acest vector și apelați metoda `toString()` pentru elementele sale. Ce observați?
5. (**2p**) Downcasting.
   - Adăugați clasei `Triangle` metoda `printTriangleDimensions` și clasei `Circle` metoda `printCircleDimensions`. Implementarea metodelor constă în afișarea bazei si inaltimii respectiv razei.
   - Parcurgeți vectorul de la exercițiul anterior și, folosind downcasting la clasa corespunzătoare, apelați metodele specifice fiecărei clase (`printTriangleDimensions` pentru `Triangle` și `printCircleDimensions` pentru `Circle`). Pentru a stabili tipul obiectului curent folosiți operatorul **`instanceof`**.
   - Modificați programul anterior astfel încât downcast-ul să se facă mereu la clasa `Triangle`. Ce observați?
   - Modificați programul anterior astfel încât să nu mai aveți nevoie de `instanceof` deloc.

Exercițiul 6 este independent de cele de mai sus

1. (**1.5p + 1.5p**) Implementați două clase `QueueAggregation` și `QueueInheritance` pe baza clasei `Array` furnizate de noi, utilizând, pe rând, ambele abordări: **moștenire** și **agregare**. Precizări:
   - Coada va conține elemente de tip `int`.
   - Clasele `QueueAggregation` și `QueueInheritance` trebuie să ofere metodele `enqueue` și `dequeue`, specifice acestei structuri de date.
   - Clasa `Array` reprezintă un wrapper pentru lucrul cu vectori. Metoda `get(pos)` întoarce valoarea din vector de la poziția `pos`, în timp ce metoda `set(pos, val)` atribuie poziției `pos` din vector valoarea `val`. Noutatea constă în verificarea poziției furnizate. În cazul în care aceasta nu se încadrează în intervalul valid de indici, ambele metode întorc constanta ERROR definită în clasa.
   - Metoda `main` definită în clasa `Array` conține exemple de utilizare a acestei clase. Experimentați!
   - Metoda `enqueue` va oferi posibilitatea introducerii unui număr întreg în capătul cozii (dacă aceasta nu este deja plină), în timp ce metoda `dequeue` va înlătura elementul din vârful cozii și îl va întoarce (dacă coada nu este goală). În caz de insucces (coada plină la `enqueue`, respectiv goală la `dequeue`), ambele metode vor întoarce constanta ERROR.
   - Ce problemă poate apărea din cauza constantei ERROR? (Hint: Dacă în coadă am un element egal cu valoarea constantei ERROR?) Gândiți-vă la o rezolvare.
   - Ce puteți spune despre vizibilitatea metodelor `get` și `set`, în clasele `QueueAggregation` și `QueueInheritance`, în varianta ce utilizează moștenire? Ce problemă indică răspunsul? Furnizați o soluție la această problemă.

Resurse

- Arhiva zip cu clasa Array.java
- PDF laborator
- **Soluție**

Referințe

- Inheritance JavaDoc
- Upcasting and Downcasting