Agregare si Mostenire

De la POO

Salt la: Navigare, căutare

In Java, **reutilizarea** codului se face la nivel de **clasa** si se poate realiza in 2 moduri:

- **Agregare** (compunere)
- **Mostenire**

Cuprins

[ascunde] [ascunde]

Agregare (*Composition*)

Numita si **compunere**, agregarea reprezinta pur si simplu prezenta unei referinte la un obiect intr-o alta clasa. Acea clasa practic va refolosi codul din clasa corespunzatoare obiectului. Exemplu:

```
class Page {
    private String content;
    public int no;

    public Page(String c, int no) {
       this.content = c;
       this.no      = no;
    }
}

class Book {
    private Page[] pages;

    public Book(int dim, String title) {
       pages = new Page[dim];
       for (int i=0; i < dim; i++)
           pages[i] = new Page("Pagina " + i, i);
    }
}
```

Din punct de vedere conceptual, exista 2 **tipuri de agregare**:

- ***strong*** – la disparitia obiectelor continute prin agregare, existenta obiectului container inceteaza (de exemplu, o carte nu poate exista fara pagini)
- ***weak*** – obiectul-container poate exista si in absenta obiectelor agregate (de exemplu, o biblioteca poate exista si fara carti)

Initializarea obiectelor continute poate fi facuta in 3 momente de timp distincte:

- la **definirea** obiectului (inaintea constructorului: folosind fie o valoare initiala, fie blocuri de initializare)
- in cadrul **constructorului**
- chiar **inainte de folosire** (acest mecanism se numeste *lazy initialization*)

Mostenire (*Inheritance*)

Numita si **derivare**, mostenirea este un mecanism de refolosire a codului specific limbajelor orientate obiect si reprezinta posibilitatea de a defini o clasa care **extinde** o alta clasa deja existenta. Ideea de baza este de a **prelua** functionalitatea existenta intr-o clasa si de a **adauga** una noua sau de a o **modela** pe cea existenta.

Clasa existenta este numita **clasa-parinte**, **clasa de baza** sau **super-clasa**. Clasa care extinde clasa-parinte se numeste **clasa-**

copil (child), **clasa derivata** sau **sub-clasa**. Pentru detalii si implementare in Java va rugam sa cititi un document pe acest subiect aflat aici (documentul este in limba engleza).

Agregare vs. mostenire

Cand se foloseste mostenirea si cand compunerea?

Raspunsul la aceasta intrebare depinde, in principal, de datele problemei analizate dar si de conceptia designerului, neexistand o reteta general valabila in acest sens. In general, **compunerea** este folosita atunci cand se doreste folosirea **trasaturilor** unei clase in interiorul altei clase, dar **nu** si interfata sa (prin mostenire, noua clasa ar expune si metodele clasei de baza). Putem distinge doua cazuri largi:

- uneori se doreste implementarea functionalitatii obiectului continut in noua clasa si **limitarea** actiunilor utilizatorului la metodele din noua clasa (mai exact, se doreste sa nu se permita utilizatorului folosirea metodelor din vechea clasa). Pentru a obtine acest efect se va **agrega** in noua clasa un obiect de tipul clasei continute si avand specificatorul de acces `private`.
- obiectul continut (agregat) trebuie/se doreste a fi accesat **direct**. In acest caz vom folosi specificatorul de acces `public`. Un exemplu in acest sens ar fi o clasa numita `Masina` care contine ca membrii publici obiecte de tip `Motor`, `Roata` etc (vezi exemplul din Bruce Eckel).

Mostenirea este un mecanism care permite crearea unor versiuni "specializate" ale unor clase existente (de baza). Mostenirea este folosita in general atunci cand se doreste construirea unui tip de date care sa reprezinte o implementare specifica (o specializare oferita prin clasa derivata) a unui lucru mai general. Un exemplu simplu ar fi clasa `Dacia` care mosteneste clasa `Masina`.

Diferenta dintre mostenire si agregare este de fapt diferenta dintre cele 2 tipuri de relatii majore prezente intre obiectele unei aplicatii :

- **is a** - indica faptul ca o clasa este derivata dintr-o clasa de baza (intuitiv, daca avem o clasa `Animal` si o clasa `Caine`, atunci ar fi normal sa avem `Caine` derivat din `Animal`, cu alte cuvinte `Caine` *is a* `Animal`)
- **has a** - indica faptul ca o clasa-container are o clasa continuta in ea (intuitiv, daca avem o clasa `Masina` si o clasa `Motor`, atunci ar fi normal sa avem `Motor` referit in cadrul `Masina`, cu alte cuvinte `Masina` *has a* `Motor`)

Upcasting si Downcasting

Convertirea unei referinte la o clasa derivata intr-una a unei clase de baza ale acesteia aceasta poarta numele de **upcasting**. Upcasting-ul este facut **automat** si **nu** trebuie declarat explicit de catre 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 automat
    }
}
```

Desi obiectul `flute` este o instanta a clasei `Wind`, acesta este pasat ca parametru in 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 (in special UML) in care mostenirea se reprezinta prin 2 blocuri asezate unul sub altul, reprezentand cele 2 clase (sus este clasa de baza iar jos clasa derivata), unite printr-o sageata orientata spre clasa de baza.

Downcasting este operatia **inversa** upcast-ului si este o conversie explicita de tip in care se merge in **jos** pe ierarhia claselor (se converteste o clasa de baza intr-una derivata). Acest cast trebuie facut **explicit** de catre programator. Downcasting-ul este **posibil** numai daca obiectul declarat ca fiind de o clasa de baza este, de fapt, instanta clasei derivate catre care se face downcasting-ul.

Iata un exemplu in 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[0] = new Wolf();
        a[1] = new Snake();

        for (int i = 0; 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
        }
    }
}
```

In liniile marcate cu **2** si **3** se executa un downcast de la `Animal` la `Wolf`, respectiv `Snake` pentru a putea fi apelate metodele specifice definite in aceste clase. Inaintea executiei downcast-ului (conversia de tip la Wolf respectiv Snake) verificam daca obiectul respectiv este de tipul dorit (utilizand operatorul **`instanceof`**). Daca am incerca sa facem downcast catre tipul `Wolf` al unui obiect instantiat la `Snake` masina virtuala ar semnala acest lucru aruncand o exceptie la rularea programului.

Apelarea metodei `eat()` (linia 1) se face direct, fara downcast, deoarece aceasta metoda este definita si in clasa de baza. Datorita faptului ca `Wolf` suprascrie (*overrides*) metoda `eat()`, apelul `a[0].eat()` va afisa "Wolf eating". Apelul `a[1].eat()` va apela metoda din clasa de baza (la iesire va fi afisat "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 raspunsul la intrebarea: *este nevoie de mostenire?* este dat de raspunsul la intrebarea: *am nevoie de upcasting?* Aceasta deoarece upcasting-ul se face atunci cand pentru unul sau mai multe obiecte din clase derivate se executa aceeasi metoda definita in clasa parinte.

Implicatii ale mostenirii

- specificatorul de acces `protected`
  - folosit in declararea unui camp sau a unei metode dintr-o clasa - specifica faptul ca membrul sau metoda respectiva poate fi accesata doar din cadrul **clasei** insesi sau din clasele **derivate** din aceasta clasa.
- specificatorul de acces `private`
  - folosit in declararea unui camp sau a unei metode dintr-o clasa - specifica faptul ca membrul sau metoda respectiva poate fi accesata doar din cadrul **clasei** insesi, **nu** si din clasele derivate din aceasta clasa.
- cuvantul cheie `final`
  - folosit la declararea unei metode, implicand faptul ca metoda nu poate fi suprascrisa in clasele derivate
  - folosit la declararea unei clase, implicand faptul ca acea clasa nu poate fi derivata

Pe langa reutilizarea codului, mostenirea da posibilitatea de a dezvolta pas cu pas o aplicatie (procedeul poarta numele de *incremental development*). Astfel, putem folosi un cod deja functional si adauga alt cod nou la acesta, in felul acesta izolandu-se bug-urile in codul nou adaugat. Pentru mai multe informatii cititi capitolul *Reusing Classes* din cartea *Thinking in Java (Bruce Eckel)*

Exercitii

1. (**2p**) Intrucat in ierarhia de clase Java, clasa `Object` se afla in radacina arborelui de mostenire pentru orice clasa, orice clasa va avea acces la o serie de facilitati oferite de `Object`. Una dintre ele este metoda `toString()`, al carei scop este de a oferi o reprezentare a unei instante de clasa sub forma unui sir de caractere.
   - Definiti clasa **`Persoana`** cu un membru `nume` de tip `String`, si o metoda `toString()` care va returna acest nume.
   - Clasa va avea, de asemenea:
     - un constructor fara parametri
     - un constructor ce va initializa numele.
   - Din ea derivati clasele **`Profesor`** si **`Student`**:
     - Clasa `Profesor` va avea membrul `materie` de tip `String`.

        - Clasa `Student` va avea membrul `nota` de tip `int`.
        - Clasele vor avea:
            - constructori fara parametri
            - constructori care permit initializarea membrilor. Identificati o modalitate de **reutilizare** a codului existent.
    - Instantiati clasele `Profesor` si `Student`, si apelati metoda `toString()` pentru fiecare instanta.
2. (**2p**) Adaugati metode `toString()` in cele doua clase derivate, care sa returneze tipul obiectului, numele si membrul specific. De exemplu:
    - pentru clasa `Profesor`, se va afisa: "Profesor: Ionescu, POO"
    - pentru clasa `Student`, se va afisa: "Student: Popescu, 5"

    Modificati implementarea `toString()` din clasele derivate astfel incat aceasta sa utilizeze implementarea metodei `toString()` din clasa de baza.
3. (**1p**) Adaugati o metoda `equals` in clasa `Student`. Justificati criteriul de echivalenta ales.
4. (**1p**) Upcasting.
    - Creati un vector de obiecte `Persoana` si populati-l cu obiecte de tip `Profesor` si `Student` (upcasting).
    - Parcurgeti acest vector si apelati metoda `toString()` pentru elementele sale. Ce observati?
5. (**2p**) Downcasting.
    - Adaugati clasei `Profesor` metoda `preda` si clasei `Student` metoda `invata`. Implementarea metodelor consta in afisarea numelui si a actiunii.
    - Parcurgeti vectorul de la exercitiul anterior si, folosind downcasting la clasa corespunzatoare, apelati metodele specifice fiecarei clase (`preda` pentru `Profesor` si `invata` pentru `Student`). Pentru a stabili tipul obiectului curent folositi operatorul **`instanceof`**.
    - Modificati programul anterior astfel incat downcast-ul sa se faca mereu la clasa `Profesor`. Ce observati?
6. (**2.5p + 2.5p**) Implementati o clasa `Stiva`, pe baza clasei `Array` furnizate de noi, utilizand, pe rand, ambele abordari: **mostenire** si **agregare**. Precizari:
    - Stiva va contine elemente de tip `int`.
    - Clasa `Stiva` trebuie sa ofere metodele `push` si `pop`, specifice acestei structuri de date.
    - Clasa `Array` reprezinta un wrapper pentru lucrul cu vectori. Metoda `get(pos)` intoarce valoarea din vector de la pozitia `pos`, in timp ce metoda `set(pos, val)` atribuie pozitiei `pos` din vector valoarea `val`. Noutatea consta in verificarea pozitiei furnizate. In cazul in care aceasta nu se incadreaza in intervalul valid de indici, ambele metode intorc constanta `ERROR` definita in clasa.
    - Metoda `main` definita in clasa `Array` contine exemple de utilizare a acestei clase. Experimentati!
    - Metoda `push` va oferi posibilitatea introducerii unui numar intreg in varful stivei (daca aceasta nu este deja plina), in timp ce metoda `pop` va inlatura elementul din varful stivei si il va intoarce (daca stiva nu este goala). In caz de insucces (stiva plina la `push`, respectiv goala la `pop`), ambele metode vor intoarce constanta `ERROR`.
    - Ce puteti spune despre vizibilitatea metodelor `get` si `set`, in clasa `Stiva`, in varianta ce utilizeaza mostenire? Ce problema indica raspunsul? Furnizati o solutie la aceasta problema.

- **Solutii**

Trusa de unelte

- Ce se leagă aici
- Modificări corelate
- Trimite fişier
- Pagini speciale
- Versiune de tipărit
- Legătură permanentă
- Print as PDF

- Ultima modificare 17:19, 18 octombrie 2011.
- Această pagină a fost vizitată de 8.693 ori.
- Conţinutul este disponibil sub GNU Free Documentation License 1.2.
- Politica de confidenţialitate
- Despre POO
- Termeni