Laboratorul 3: Agregare și moștenire

Video introductiv: link [https://www.youtube.com/watch?v=Gb-p4tMrdBM]

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 corespunzătoare obiectului.

- **Agregarea** (aggregation) - obiectul-container poate exista și în absența obiectelor agregate, de aceea este considerată o *asociere slabă* (*weak association*). În exemplul de mai jos, 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 jos, 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*))

Exemple de cod:

Compunere:

```
public class Foo {
    // Obiectul Bar nu poate exista dacă obiectul Foo nu există
    private Bar bar = new Bar();
}
```

Agregare:

```
public class Foo {
    private Bar bar;

    // Obiectul Bar poate continua să existe chiar dacă obiectul Foo nu există
    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;                       // Compunere
    private Page[] pages;                       // Compunere
    private LibraryRow libraryRow = null;       // Agregare

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

        pages = new Page[size];

        for (int i = 0; i < size; i++) {
           pages[i] = new Page("Page " + i, i);
        }
    }
}

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

    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);
```

```
        // După ce nu mai există nici o referință la obiectul Carte,
        // Garbage Collector-ul va șterge (la un moment dat, nu
        // neapărat imediat) acea instanță, dar obiectul LibraryRow
        // transmis constructorului nu este afectat.

        book = null;
    }
}
```

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 [https://en.wikipedia.org/wiki/Multiple_inheritance#The_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`.

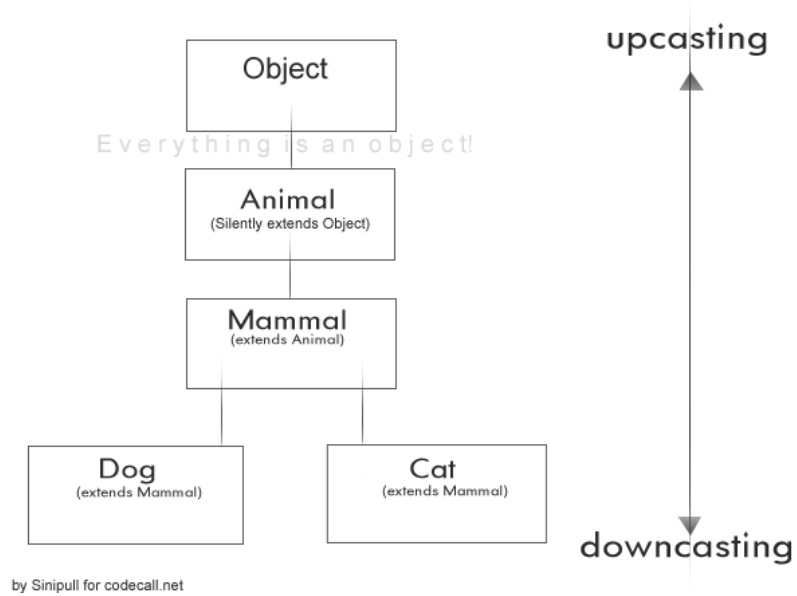
Diferența dintre moștenire și agregare este de fapt diferența dintre cele 2 tipuri de relații majore prezente între obiectele unei aplicații :

- **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();
    }
}

// Obiectele Wind sunt instrumente
// deoarece au aceeași interfață:
public class Wind extends Instrument {
    public static void main(String[] args) {
        Wind flute = new Wind();
        Instrument.tune(flute); // !! Upcasting automat pentru că metoda primește
                                // un obiect de tip Instrument, nu un obiect de tip Wind
                                // Deci ar fi redundant să faci un cast explicit cum ar fi:
                                // 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 [http://en.wikipedia.org/wiki/Unified_Modeling_Language]) 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[0] = new Wolf();    // Upcasting automat
        a[1] = new Snake();   // Upcasting automat

        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
            }
```

```
        }
    }
}
```

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 ilustrat 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() {
        // avem nevoie de această metodă deoarece vom crea un vector
        // cu instanțe Animal și vom apela această metodă pe ele
    }
}

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

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

```
    }
}

class Snake extends Animal {
    public void action() {
        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();

            // acum că ele sunt numite la fel, putem apela metoda action
            // din clasa Animal (observați de ce a fost nevoie să definim
            // metoda action în clasa Animal), iar metoda corespunzătoare
            // va fi apelată pentru tipul specific al instanței a[i]

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

Codul va afișa:

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

De ce este instanceof considerat bad practice?

- face codul repetitiv - downcasting și apelare de metodă la fiecare branch de if/else
- face codul mai greu de întreținut pe termen lung - codul principal trebuie updatat de fiecare dată când se introduce o nouă subclasă (vezi exemplul de mai sus - introducerea unui noi clase derivate din Animal va determina un alt branch de if/else)
- face codul mai dificil de citit - este mult mai ușor să ne uităm la metodele suprascrise cu tag-ul @Override decât să căutam fiecare metoda cu un nume diferit pe rând
- distruge designul orientat-obiect din perspectiva polimorfismului
- încalcă open-closed principle [https://www.baeldung.com/java-open-closed-principle]
- nu poate fi folosit cu tipuri generice, `instanceof` fiind un **type comparison operator** ce va compara la runtime, în schimb genericitatea fiind rezolvată la compile-time. Mai multe detalii aici [https://www.baeldung.com/java-instanceof#generics] si in laboratorul 11 [https://ocw.cs.pub.ro/courses/poo-ca-cd/laboratoare/genericitate] despre genericitate.

Soluții pentru evitarea `instanceof`:

- polimorfism
- Visitor [https://ocw.cs.pub.ro/courses/poo-ca-cd/laboratoare/visitor]

Când este necesar să folosim `instanceof`?

`instanceof` poate fi folosit când nu controlăm ierarhia claselor și suntem obligați să facem testarea tipului deoarece nu putem aplica polimorfismul - clasele provin dintr-o librărie externă.

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).

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");
    }

    // Exemplu de suprascriere
    public void addGasoline() {
        // do something
    }

    // Exemplu de supraîncărcare
    public void addGasoline(Integer gallons) {
        // do something
    }
}
```

Metodele dependente de instanță sunt polimorfice (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(); // afișează Car
b.print(); // afișează Dacia
c.print(); // afișează Dacia
d.print(); // aruncă 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(); // afișează Car
b.print(); // afișează Car pentru că tipul dat la inițializare al lui b este Car
c.print(); // afișează Dacia pentru că tipul dat la inițializare al lui c este  Dacia
d.print(): // afișează Car pentru că tipul dat la inițializare al lui d este 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.

Suprascrierea corecta a metodei equals(Object o)

Una din problemele cele mai des întâlnite este suprascrierea corectă a metodei *equals*. Mai jos putem vedea un exemplu de suprascriere incorectă a acestei metode.

```
public class Car {
    public boolean equals(Car c) {
        System.out.println("Car");
        return true;
    }

    public boolean equals(Object o) {
        System.out.println("Object");
        return false;
    }
}
```

Prima metodă este o **supraîncărcare** a metodei equals iar a doua metodă este **suprascrierea** metodei equals.

```
Car a = new Car();
Dacia b = new Dacia();
Int c = new Int(10);

a.equals(a); // afișează Car
a.equals(b); // afișează Car deoarece se face upcasting de la Dacia la Car
a.equals(c); // afișează Object deoarece se face upcasting de la Int la Object
```

Problema care se poate observa este că putem pasa ca argumente metodei equals si tipuri de date diferite de Car, lucru ce ar putea arunca excepții de cast sau când vrem să accesăm anumite proprietăți din instanță. Mai jos este modul corect de suprascrie metoda equals.

```
public class Car {
    public boolean equals(Car c)
    {
        return true;
    }

    public boolean equals(Object o)
    {
        if (o == this) {
            return true;
        }

        if (!(o instanceof Car)) {
            return false;
        }

        return equals((Car) o);
```

```
    }
}
```

De reținut că folosirea `instanceof` nu este recomandată, însă în acest caz este singurul mod prin care ne putem asigura ca instanța de obiect trimisă metodei este de tip `Car`.

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();  // apelează metoda părinte

        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();    // apelează constructorul părinte
                    // acest apel trebuie să fie pe prima linie a constructorului !!

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

    public Subclass(int a) {
        super(a);   // apelează constructorul părinte
                    // acest apel trebuie să fie pe prima linie a constructorului !!

        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

Utilizarea clasei ArrayList. Exemple

Clasa ArrayList este un array redimensionabil, iar această clasă poate fi găsită în pachetul `java.util`. Principalele metode ale acestei clase sunt:

- `add()` - adaugă un element la ArrayList.
- `get()` - accesează elementul de pe o anumită poziție din ArrayList.
- `size()` - returnează numărul de elemente din ArrayList.
- `set()` - modifica un element de pe o anumită poziție din ArrayList.

- `remove()` - șterge un element de pe o anumită poziție din ArrayList.
- `clear()` - șterge toate elementele din ArrayList.

Pentru a putea parcurge elementele unui ArrayList se poate folosit atât un for-each, cât și clasa Iterator. Un exemplu de utilizare a metodelor clasei ArrayList și de parcurgere a elementelor stocate de această clasă este următorul:

```
import java.util.ArrayList;
import java.util.Iterator;

public class Main {
    public static void main(String[] args) {

        ArrayList<String> animals = new ArrayList<>();
        animals.add("Dog");
        animals.add("Cat");
        animals.add("Sheep");

        System.out.println("First animal from the list is: " + animals.get(0));
        System.out.println("Number of animals from the list: " + animals.size());

        animals.set(0, "Lion");
        System.out.println("First animal from the list is: " + animals.get(0));

        animals.remove(0);

        for (String animal : animals) {
            System.out.println(animal);
        }

        Iterator iterator = animals.iterator();
        while(iterator.hasNext()) {
            System.out.println(iterator.next());
        }

        animals.clear();
    }
}
```

Summary

Relații între obiecte:

- Agregare - **has a**
- Moștenire - **is a**

Upcasting

- convertire **copil** ⇒ **părinte**
- realizată automat

Downcasting

- convertire **părinte ⇒copil**
- trebuie făcută explicit de către programator
- încercați să evitați folosirea operatorului **instanceof**

Suprascrierea

- înlocuirea funcționalității metodei din clasa de bază în clasa derivată
- păstreaza numele și semnătura metodei

Supraincărcarea

- în interiorul clasei pot exista mai multe metode cu același nume, cu condiția ca semnătura (tipul, argumentele) să fie diferită

super

- instanța clasei părinte
- amintiți-vă din laboratorul anterior că **this** se referă la instanța clasei curente

Exerciții

Pentru acest laborator este de recomandat să creați un pachet numit `lab3`, unde veți face subpachete pentru task-uri.

Gigel vrea să-i facă mamei sale un cadou de ziua ei și știe că-i plac foarte mult bomboanele. El are nevoie de ajutorul vostru pentru a construi cel mai frumos și gustos cadou:

Task 1 [2p]

Veți proiecta o clasă `CandyBox`, care va conține câmpurile private `flavor` (String) și `origin` (String). Clasa va avea, de asemenea:

- un constructor fără parametri
- un constructor ce va inițializa toate câmpurile
- o metodă de tip float `getVolume()`, care va întoarce valoarea 0;
- Întrucât clasa `Object` se află în rădăcina arborelui de moștenire pentru orice clasă, orice instanță 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 unei instanțe sub forma unui șir de caractere, utilizată în momentul apelului System.out.println(). Adăugați o metodă `toString()`, care va returna flavor-ul și regiunea de proveniență a cutiei de bomboane.

Task 2 [2p]

Din ea derivați clasele `Lindt`, `Baravelli`, `ChocAmor`. Pentru un design interesant, cutiile vor avea forme diferite:

- *Lindt* va conține `length`, `width`, `height` (float);
- *Baravelli* va fi un cilindru. Acesta va conține un câmp `radius` și unul `height` (float);
- *ChocAmor*, fiind un cub, va conține un camp `length` (float);

Clasele vor avea:

- constructori fără parametri
- constructori care permit inițializarea membrilor. Identificați o modalitate de reutilizare a codului existent. Pentru fiecare tip de cutie veți inițializa, în constructor, câmpurile `flavor` și `origin` cu tipul corespunzător
- Suprascrieți metoda *getVolume()* pentru a întoarce volumul specific fiecărei cutii de bomboane, în funcție de tipul său.
- Suprascrieți metoda *toString()* în clasele derivate, astfel încât aceasta să utilizeze implementarea metodei toString() din clasa de bază. Returnați un mesaj de forma *"The " + origin + " " + flavor + " has volume " + volume*;

Task 3 [1p]

Adăugați o metodă `equals()` în clasa `CandyBox`. Justificați criteriul de echivalentă ales. Vedeți metodele clasei Object [https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/lang/Object.html], moștenită de toate clasele - Object are metoda equals, a cărei implementare verifică echivalența obiectelor comparând referințele. Creați obiecte de tip Lindt și testați egalitatea lor.

Hint: Puteți genera automat metoda, cu ajutorul IDE. Selectați câmpurile considerate și analizați în ce fel va fi suprascrisă metoda equals.

Task 4 - *`Upcasting`* [2p]

Acum că am stabilit tipul cutiilor de bomboane, putem construi cadoul, rămânând la latitudinea voastră care va fi designul lui. În pachetul java.util se găsește clasa `ArrayList`, care definește un resizable array, cu metodele specifice (add, size, get, lista lor completa este în documentatie [https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/util/ArrayList.html]). Creați o clasă `CandyBag`, care va conține un ArrayList cu mai multe cutii din fiecare tip.

Task 5 - *`Downcasting`* [1p]

Adăugați clasei `Baravelli`, funcția `printBaravelliDim()`, care va afișa dimensiunile razei și înălțimii. În mod analog, procedați cu celelalte tipuri de cutii, adaugând metodele `printChocAmorDim()` și `printLindtDim()`, în care să afișați dimensiunile fiecărei cutii.

Task 6 - *Agregare* [2p]

Gigel va vrea sa trimită prin curier cadoul, pentru a nu-l găsi mama lui mai devreme. Ajutați-l să determine locația, creând clasa "Area", care va conține un obiect de tip `CandyBag`, un câmp "number" (int) și un câmp "street" (String) Clasa va avea, de asemenea:

- un constructor fără parametri
- un constructor ce va inițializa toate câmpurile
- Acum ca am finalizat construcția, îi vom oferi mamei informații despre cadoul ei printr-o felicitare. Creați o metoda `getBirthdayCard()`, care va afișa, pe primul rand, adresa completă, iar apoi un mesaj de `la multi ani`.
- Tot aici, parcurgeți array-ul, apelând metoda `toString()` pentru elementele sale.

- Parcurgeți array-ul și, folosind downcasting la clasa corespunzătoare, apelați metodele specifice fiecărei clase. Pentru a stabili tipul obiectului curent folosiți operatorul `instanceof`
- În final, modificați cum considerați programul anterior astfel încât să nu mai aveți nevoie de instanceof.

Resurse

- Exerciții din alți ani

Referințe

- UML Diagrams [https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/]
- Aggregation vs Composition [https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-aggregation-vs-composition/]
- Inheritance JavaDoc [http://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html]
- Multiple Inheritance [https://www.journaldev.com/1775/multiple-inheritance-in-java]
- Upcasting and Downcasting [http://forum.codecall.net/topic/50451-upcasting-downcasting/]

poo-ca-cd/laboratoare/agregare-mostenire.txt · Last modified: 2021/10/24 22:49 by florin.mihalache