Laboratorul 6: Clase interne

Video introductiv: link [https://youtu.be/KIyaRwog3q8]

Slides din video: link

Obiective

- prezentarea conceptului de clasă internă
- exemplificarea tipurilor de clase interne folosind Java
- utilizarea claselor interne, în special ale celor anonime
- utilizarea funcțiilor lambda

Introducere

Clasele declarate în interiorul unei alte clase se numesc clase interne (*nested classes*). Acestea permit gruparea claselor care sunt legate logic și controlul vizibilității uneia din cadrul celorlalte.

Clasele interne sunt de mai multe tipuri:

- clase interne normale (*regular inner classes*)
- clase anonime (*anonymous inner classes*)
- clase interne statice (*static nested classes*)
- clase interne metodelor (*method-local inner classes*) sau blocurilor

- O clasă internă se comportă ca un membru al clasei în care a fost declarată
- O clasă internă are acces la toți membrii clasei în care a fost declarată, inclusiv cei **private**
- O clasă internă poate fi `public`, `final`, `abstract` dar și `private`, `protected` și `static`, însumând modificatorii claselor obișnuite și cei permiși metodelor și variabilelor
- Nested classes vs. Inner classes [https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html]

Clase interne "normale"

O clasă internă este definită în interiorul unei clase și poate fi accesată doar la runtime printr-o instanță a clasei externe (la fel ca metodele și variabilele ne-statice). Compilatorul creează fișiere *.class* separate pentru fiecare clasă internă, în exemplul de mai jos generând fișierele `Car.class` și `Car$OttoEngine.class`, însă execuția fișierului `Car$OttoEngine.class` nu este permisă.

Test.java

```
interface Engine {
    public int getFuelCapacity();
}

class Car {
    class OttoEngine implements Engine {
        private int fuelCapacity;

        public OttoEngine(int fuelCapacity) {
            this.fuelCapacity = fuelCapacity;
        }

        public int getFuelCapacity() {
            return fuelCapacity;
        }
    }

    public OttoEngine getEngine() {
        OttoEngine engine = new OttoEngine(11);
        return engine;
    }
}

public class Test {
    public static void main(String[] args) {
        Car car = new Car();

        Car.OttoEngine firstEngine = car.getEngine();
        Car.OttoEngine secondEngine = car.new OttoEngine(10);

        System.out.println(firstEngine.getFuelCapacity());
        System.out.println(secondEngine.getFuelCapacity());
```

```
    }
}
```

```
student@poo:~$ javac Test.java
student@poo:~$ ls
Car.class Car$OttoEngine.class Engine.class Test.class Test.java
```

Urmăriți exemplul de folosire a claselor interne de mai sus. Adresați-vă asistentului pentru eventuale neclarități.

Dintr-o clasă internă putem accesa **referința la clasa externă** (în cazul nostru `Car`) folosind numele acesteia și keyword-ul *this*:

```
Car.this;
```

Modificatorii de acces pentru clase interne

Așa cum s-a menționat și în secțiunea Introducere, claselor interne le pot fi asociați **orice** identificatori de acces, spre deosebire de clasele `top-level` Java, care pot fi doar `public` sau `package-private`. Ca urmare, clasele interne pot fi, în plus, `private` și `protected`, aceasta fiind o modalitate de a **ascunde implementarea**.

Folosind exemplul anterior, dacă modificăm clasa `OttoEngine` pentru a fi privată apar erori de compilare.

- Tipul `Car.OttoEngine` nu mai poate fi accesat din exterior. Acest neajuns poate fi rezolvat cu ajutorul interfeței `Engine`. Asociindu-i clasei interne `Car.OttoEngine` supertipul `Engine` prin moștenire, putem instanția clasa prin **upcasting**.
- Fiind privată, clasa internă are implicit toți constructorii privați. Totuși, putem instanția obiecte de tipul `Car.OttoEngine` în interiorul clasei `Car`, urmând să le întoarcem folosind tot upcasting la `Engine`. Astfel, folosind metode getEngine, ascundem complet implementarea clasei `Car.OttoEngine`.

Clase anonime

În dezvoltarea software, există situații când o componentă a aplicației are o utilitate suficient de mare pentru a putea fi considerată o entitate separată (sau clasă). De multe ori, însă, aceasta nu este utilizată decât într-o porțiune restrânsă din aplicație, într-un context foarte specific (într-un lanț de moșteniri sau ierarhie de interfețe).

Putem folosi **clase interne anonime** în locul definirii unei clase cu număr de utilizări reduse. Acestea nu au nume și apar în program ca instanțe ale unei clase moștenite (sau a unei interfețe extinse), care suprascriu (sau implementează) anumite metode.

Întorcându-ne la exemplul cu clasa top-level `Car`, putem rescrie metoda `getEngine()` a acesteia astfel:

```
[...]
class Car {
    public Engine getEngine(int fuelCapacity) {
        return new Engine () {
            private int fuelCapacity = 11;

            public int getFuelCapacity() {
                return fuelCapacity;
            }
        };
    }
}
[...]
```

Metoda folosită mai sus elimină necesitatea creări unei clase interne “normale”, reducând volumul codului și crescând lizibilitatea acestuia. Sintaxa `return new Engine() { … }` se poate citi astfel: ***“Crează o clasă care implementează interfața Engine, conform următoarei implementări”.***

Observații:

- Obiectul este instanțiat imediat după `return`, folosind `new` (referința întoarsă de `new` va fi `upcast` la tipul de bază: `Engine`)
- Numele clasei instanțiate este absent (ea este *anonimă*), însă ea este de `tipul Engine`, prin urmare, va implementa metoda/metodele din interfață (cum e metoda `getFuelCapacity`). Corpul clasei urmează imediat instanțierii.

Limitări

- Clasele anonime **nu pot avea constructori** din cauză că nu au nume (nu am ști cum să numim constructorii). Această restricție asupra claselor anonime ridică o problemă: în mod implicit, clasă de bază este creată cu constructorul *default*.
- O clasă internă anonimă poate extinde o clasă *sau* să implementeze o singură interfață, nu poate face pe ambele împreună ca la clasele ne-anonime (interne sau nu), și nici nu poate să implementeze mai multe interfețe.

Clasele interne anonime declarate în metode pot folosi variabilele declarate în metoda respectivă și parametrii metodei dacă aceștia sunt *final* sau ***effectively final***. Dacă o variabilă nu e declarată final dar nu se modifică după inițializare, atunci este effectively final

[https://docs.oracle.com/javase/tutorial/java/javaOO/localclasses.html#accessing-members-of-an-enclosing-class].

Variabilele si parametrii metodelor se află pe segmentul de stivă din memorie creat pentru metoda respectivă, ceea ce face ca ele să nu existe la fel de mult cât clasa internă. Dacă variabila este declarată `final`, atunci la runtime se va stoca o copie a acesteia ca un câmp al clasei interne, în acest mod putând fi accesată și după execuția metodei.

Până la Java 8 [http://ilkinulas.github.io/programming/java/2016/03/27/effectively-final-java.html] nu exista conceptul de *effectively final* si toate aceste variabile trebuiau declarate *final*

Expresii Lambda

Când veți scrie o clasă anonimă într-un IDE cum e IntelliJ, veți primi un warning care vă recomandă să o transformați într-o lambda. Aceasta recomandare este valabilă doar pentru implementarea claselor/interfețelor cu o singură metoda.

Lambda este un concept din programarea funcțională (o să îl învățați în semestrul 2 la Paradigme de Programare) și reprezintă o funcție anonimă. Majoritatea limbajelor de nivel înalt au introdus suport pentru acest concept în ultimii 15 ani, inclusiv Java, din versiunea 8.

În exemplul următor am transformat clasa anonimă din secțiunea precedentă. Atenție! IDE-ul nu ar fi recomandat pentru cazul din acel exemplu transformarea în lambda pentru că aveam definită o variabilă private int fuelCapacity = 11, deci codul nu era echivalent unei simple implementări de funcții. Eliminând acea variabilă constantă și punând-o în metodă am putut face transformarea.

```
class Car {
    public Engine getEngine() {
        return () -> 11; // expresie lambda
    }
}
```

În codul de mai sus declararea clasei anonime și suprascrierea metodei din getFuelCapacity() a fost înlocuită cu o expresie lambda. O altă situație des intalnită de folosire a lambda-urilor este pentru transmiterea de funcții ca parametru iar api-uri precum cel de filtrare al colecțiilor le utilizează intens (exemplu [https://www.baeldung.com/java-stream-filter-lambda]).

În exemplul de mai jos aveți niște exemple simple de folosire lambda ca paremetru pentru metode. Parametrii primiți în lambda pot fi zero (*()* ca în exemplul cu fuelCapacity) sau mai mulți (param1, param2, …).

```
    public static void main(String[] args) {

        ArrayList<Integer> values = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));

        values.removeIf((v) -> v % 2 == 0);           // parametru o lambda care intoarce daca numărul primit este par sau nu

        values.forEach((v) -> System.out.println(v)); // parametru o lambda care afiseaza argumentul primit
        values.forEach(System.out::println);          // referinta catre metoda println
    }
}
```

Operatorul :: [https://www.geeksforgeeks.org/double-colon-operator-in-java/] este folosit pentru referințierea metodelor.

O să mai oferim exemple și detalii despre lambdas și în laboratoarele următoare, de exemplu cel cu Colecții sau cel pentru concepte din Java 8.

Clase interne statice

În secțiunile precedente, s-a discutat despre clase interne ale caror instanțe există doar în contextul unei instanțe a clasei exterioare, astfel că pot accesa membrii obiectului exterior direct.

Clasele interne le vedem ca pe membri ai claselor exterioare, așa cum sunt câmpurile și metodele. De accea, ele pot lua toți modificatorii disponibili membrilor, printre ei aflându-se modificatori pe care clasele exterioare nu le pot avea (e.g. private, static).

Așa cum pentru a accesa metodele și variabilele statice ale unei clase nu este nevoie de o instanță a acesteia, putem obține o referință către o clasă internă fără a avea nevoie de o instanță a clasei exterioare.

Pentru clasele interne statice:

- Nu avem nevoie de un obiect al clasei externe pentru a crea un obiect al clasei interne
- Nu putem accesa câmpuri nestatice ale clasei externe din clasă internă (nu avem o instanță a clasei externe)

*Clasele interne statice nu au nevoie de o instanță a clasei externe → atunci **de ce le facem interne** acesteia?*

- Pentru a grupa clasele, dacă o clasă internă statică A.B este folosită doar de A, atunci nu are rost să o facem top-level.

*Avem o clasă internă MyOuterClass.MyInnerClass, **de ce să o facem statică**?*

- Dacă în interiorul clasei MyInnerClass nu avem nevoie de nimic specific instanței clasei externe MyOuterClass, deci nu avem nevoie de o instanță a acesteia o putem face statică

Clase interne în metode și blocuri

Primele exemple prezintă modalitățile cele mai uzuale de folosire a claselor interne. Totuși, design-ul claselor interne este destul de complex și exista modalitati mai “obscure” de a le folosi: clasele interne pot fi definite și în cadrul metodelor sau al unor blocuri arbitrare de cod.

În exemplul următor, clasa internă a fost declarată în **interiorul funcției** `getEngine`. În acest mod, vizibilitatea ei a fost redusă pentru ca nu poate fi instanțiată decât în această funcție.

Singurii modificatori care pot fi aplicați acestor clase sunt `abstract` și `final` (binențeles, nu amândoi deodată).

Pentru a accesa variabile declarate în metodă respectivă sau parametri ai acesteia, ele trebuie să fie *final*. Vedeți explicația în secțiunea despre clasele anonime.

Test.java

```
[...]
class Car {
    public Engine getEngine() {
        class OttoEngine implements Engine {
            private int fuelCapacity = 11;

            public int getFuelCapacity() {
                return fuelCapacity;
            }
        }

        return new OttoEngine();
    }
}
[...]
```

Exemplu de clasa internă declarata într-un **bloc**:

```
[...]
class Car {
    public Engine getEngine(int fuelCapacity) {
        if (fuelCapacity == 11) {
            class OttoEngine implements Engine {
                private int fuelCapacity = 11;

                public int getFuelCapacity() {
                    return fuelCapacity;
                }
            }

            return new OttoEngine();
        }

        return null;
    }
}
[...]
```

În acest exemplu, clasa internă `OttoEngine` este defintă în cadrul unui bloc *if*, dar acest lucru nu înseamnă că declarația va fi luată în considerare doar la rulare, în cazul în care condiția este adevarată.

Semnificația declarării clasei într-un bloc este legată strict de vizibilitatea acesteia. La compilare clasa va fi creată indiferent care este valoarea de adevăr a condiției *if*.

Moștenirea claselor interne

Deoarece constructorul clasei interne trebuie sa se *atașeze* de un obiect al clasei exterioare, moștenirea unei clase interne este puțin mai complicată decât cea obișnuită. Problema rezidă în nevoia de a inițializa legătura (ascunsă) cu clasa exterioară, în contextul în care în clasa derivată nu mai există un obiect default pentru acest lucru.

```
class Car {
    class Engine {
        public void getFuelCapacity() {
            System.out.println("I am a generic Engine");
        }
    }
}

class OttoEngine extends Car.Engine {
    OttoEngine() {
    } // EROARE, avem nevoie de o legatura la obiectul clasei exterioare

    OttoEngine(Car car) { // OK
        car.super();
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        Car car = new Car();
        OttoEngine ottoEngine = new OttoEngine(car);
        ottoEngine.getFuelCapacity();
    }
}
```

Observăm ca `OttoEngine` moștenește doar `Car.Engine` însă sunt necesare:

- parametrul constructorului `OttoEngine` trebuie să fie de tipul clasei externe (`Car`)
- linia din constructorul `OttoEngine`: `car.super()`.

Utilizarea claselor interne

Clasele interne pot părea un mecanism greoi și uneori artificial. Ele sunt însă foarte utile în următoarele situații:

- Rezolvăm o problemă complicată și dorim să creăm o clasă care ne ajută la dezvoltarea soluției dar:
  - **nu** dorim să fie **accesibilă** din exterior sau
  - **nu** mai are **utilitate** în alte zone ale programului
- Implementăm o anumită interfață și dorim să întoarcem o referință la acea interfață, **ascunzând** în același timp implementarea.
- Dorim să folosim/extindem funcționalități ale mai **multor** clase, însă în JAVA nu putem extinde decât o singură clasă. Putem defini însă clase interioare. Acestea pot **moșteni** orice clasă și au, în plus, acces la obiectul clasei **exterioare**.
- Implementarea unei arhitecturi de control, marcată de nevoia de a trata evenimente într-un **sistem bazat pe evenimente**. Unul din cele mai importante sisteme de acest tip este **GUI** (graphical user interface). Bibliotecile Java Swing [http://en.wikipedia.org/wiki/Swing_(Java)], AWT [http://en.wikipedia.org/wiki/Abstract_Window_Toolkit], SWT [http://www.eclipse.org/swt/] sunt arhitecturi de control care folosesc intens clase interne. De exemplu, în Swing, pentru evenimente [http://docs.oracle.com/javase/tutorial/uiswing/events/index.html] cum ar fi apăsarea unui buton se poate atașa obiectului buton o tratare particulară al evenimentului de apăsare în felul următor:

```
 button.addActionListener(new ActionListener() { //interfata implementata e ActionListener
        public void actionPerformed(ActionEvent e) {
             numClicks++;
        }
    });
```

Exerciții

Schelet de laborator: Laborator6 [https://github.com/oop-pub/oop-labs/tree/master/src/lab6]

Exercițiile din acest laborator au ca scop simularea obținerii prețului unei mașini de la un dealer. Construcția obiectelor necesare o veți face de la zero conform instrucțiunilor din taskuri.

Task 1 - Structura de bază (2p)

Car Creați clasa `Car` cu următoarele proprietăți: prețul, tipul și anul fabricației.

- `Tipul` este reprezentat printr-un enum enum [https://www.w3schools.com/java/java_enums.asp] `CarType` declarat intern în Car. Acesta conține trei valori: Mercedes, Fiat și Skoda.
- `Prețul` și `anul` vor fi de tipul integers.

Creați un constructor cu toti cei trei parametri, în ordinea din enunț și suprascrieți metoda toString() pentru afișare în felul următor: Car{price=20000, carType=SKODA, year=2019}

Offer

Creați interfața `Offer` ce conține metoda: `int getDiscount(Car car);`.

Dealership

Creați clasa `Dealership` care se va ocupa cu aplicarea ofertelor pentru mașini.

Task 2 - Ofertele (4p)

În clasa `Dealership` creați trei **clase interne** private care implementează `Offer`.

- `BrandOffer` - calculează un discount în funcție de tipul mașinii:
  1. Mercedes: discount 5%;
  2. Fiat: discount 10%;
  3. Skoda: discount 15%;

- `DealerOffer` - calculează un discount în funcție de vechimea mașinii:
    1. Mercedes: discount 300 pentru fiecare an de vechime;
    2. Fiat: discount 100 pentru fiecare an de vechime;
    3. Skoda: discount 150 pentru fiecare an de vechime;
- `SpecialOffer` - calculează un discount random, cu seed 20. Generarea se va realiza în constructor utilizându-se o instanța globală a unui obiect de tip `Random` care a fost inițializat cu seed-ul **20** și cu limita superioară (bound) 1000 Random [https://docs.oracle.com/javase/8/docs/api/java/util/Random.html#Random-long-].

Adăugați o metodă în clasa `Dealership` care oferă prețul mașinii după aplicarea discount-urilor din oferte: `getFinalPrice(Car car)`

- aplicați pe obiectul `car` primit ca argument cele trei oferte in ordinea: `BrandOffer`, `DealerOffer`, `SpecialOffer`.
- metoda va returna prețul final după aplicarea ofertelor

Testare oferte: Creati 2 obiecte Car pentru fiecare tip de mașină cu urmatoarele valori:.

1. Mercedes:
    a. Pret: 20000, An: 2010;
    b. Pret: 35000, An: 2015;
2. Fiat:
    a. Pret: 3500, An: 2008;
    b. Pret: 7000, An: 2010;
3. Skoda:
    a. Pret: 12000, An: 2015;
    b. Pret: 25000, An: 2021;

- Creati un obiect de tip Dealership.
- Obțineți și afișați prețul oferit de Dealership(folosind metoda `getFinalPrice`) pentru fiecare obiect.
- De fiecare data cand se aplica o oferta asupra unui obiect de tip Car se va afisa un mesaj de tipul: “Applying x discount: y euros”, unde:
    - x reprezinta oferta care a fost aplicata(Brand, Dealer, Special, Client)
    - y reprezinta discount-ul ofertei.

Task 3 - Negocierea (2p)

Aăugați în clasa `Dealership` metoda `void negotiate(Car car, Offer offer)`. Aceasta permite clientului să propună un discount.

În metoda `main` apelați `negotiate` dând ca parametru **oferta sub formă de clasă anonimă**. Implementarea ofertei clientului reprezinta returnarea unui discount de 5%. Pentru testare folositi urmatorul obiect Car: -Pret: 20000 -Tip: Mercedes -An: 2019

Task 4 - Lambda (2p)

Testați folosirea expresiilor lambda pe următorul caz: pe o listă de obiecte de tip Car cu prețuri variate, eliminați toate mașinile care au prețul peste 25000. Afișați lista înainte și după modificare. Pentru lista folositi urmatoarele obiecte Car:

1. Mercedes:
    a. Pret: 30000, An: 2019;
    b. Pret: 50000, An: 2021;
2. Fiat:
    a. Pret: 10000, An: 2018;
3. Skoda:
    a. Pret: 20000, An: 2019;

- Hint: exemplul din secțiunea Expresii Lambda

Resurse

- Old exercises

Referințe

1. Kathy Sierra, Bert Bates. *SCJP Sun Certified Programmer for Java™ 6 - Study Guide*. Chapter 8 - Inner Classes (available online [http://firozstar.tripod.com/_darksiderg.pdf])

2. https://www.oracle.com/technetwork/java/jvmls2013kuksen-2014088.pdf [https://www.oracle.com/technetwork/java/jvmls2013kuksen-2014088.pdf]

poo-ca-cd/laboratoare/clase-interne.txt · Last modified: 2022/11/16 22:35 by daniela.becheanu