Clase interne

De la POO

Salt la: Navigare, căutare

Cuprins

[ascunde] [ascunde]

Introducere

Este posibila crearea unei clase in **interiorul** altei clase. In acest caz, prima clasa se numeste **clasa interna**. Clasele interne reprezinta o functionalitate importanta deoarece permit **gruparea** claselor care sunt legate logic si controlul **vizibilitatii** uneia din cadrul celorlalte.

Instantierea claselor interne

```
class Outer {

    class Inner {
        private int i;

        public Inner (int i) {
            this.i = i;
        }

        public int value () {
            return i;
        }
    }

    public Inner getInnerInstance () {
        Inner in = new Inner (11);
        return in;
    }
}

public class Test {

    public static void main(String[] args) {
        Outer out       = new Outer ();

        Outer.Inner in1 = out.getInnerInstance();
        Outer.Inner in2 = out.new Inner(10);

        System.out.println(in1.value());
        System.out.println(in2.value());
    }
}
```

In exemplul de mai sus avem doua modalitati de a obtine o instanta a clasei `Inner` (definita in interiorul clasei `Outer`):

- definim o metoda `getInnerInstance`, care creeaza si intoarce o astfel de instanta;
- instantiem efectiv `Inner`; observati cu atentie sintaxa folosita! Pentru a instantia `Inner`, avem nevoie de o instanta `Outer`: `out.new Inner(10);`

Modificatorii de acces pentru clase interne

Claselor interne le pot fi asociati **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, in plus, `private` si `protected`, aceasta fiind o modalitate de a **ascunde implementarea**.

```
interface Hidden {
    public int value();
}

class Outer {

    private class HiddenInner implements Hidden {
        private int i;

        public HiddenInner (int i) {
            this.i = i;
        }

        public int value () {
            return i;
        }
    }

    public Hidden getInnerInstance () {
        HiddenInner in = new HiddenInner(11);
        return in;
    }
}

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

        Outer.HiddenInner in1 = out.getInnerInstance();          // va genera eroare, tipul Outer.HiddenInner nu
este vizibil
        Outer.HiddenInner in2 = new Outer().new HiddenInner(10); // din nou eroare

        Hidden in3            = out.getInnerInstance();          // acces corect la o instanta HiddenInner
        System.out.println(in3.value());
    }
}
```

Observati definirea interfetei `Hidden`. Ea este necesara pentru a putea **asocia** clasei `HiddenInner` un *tip*, care sa ne permita folosirea instantelor acesteia. Observati, de asemenea, incercarile eronate de a instantia `HiddenInner`. Cum clasa interna a fost declarata `private`, acest tip nu mai este vizibil in exteriorul clasei `Outer`.

Varietati de clase interne

Clase interne in metode si blocuri

Primele exemple prezinta modalitatile cele mai uzuale de folosire a claselor interne. Totusi, design-ul claselor interne este destul de complet si exista modalitati mai "obscure" de a le folosi: clasele interne pot fi definite si in cadrul **metodelor** sau al unor **blocuri** arbitrare de cod.

Exemplu de clasa interna declarata intr-o **functie:**

```
interface Hidden {
    public int value ();
}

class Outer {
    public Hidden getInnerInstance() {

        class FuncInner implements Hidden {
            private int i = 11;

            public int value () {
                return i;
            }
        }

        return new FuncInner();
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        Outer out           = new Outer ();

        Outer.FuncInner in2 = out.getInnerInstance(); // EROARE: clasa FuncInner nu este vizibila
        Hidden in3          = out.getInnerInstance();

        System.out.println(in3.value());
    }
}
```

In exemplul de mai sus, clasa interna a fost declarata in **interiorul functiei** `getInnerInstance`. In acest mod, vizibilitatea ei a fost **redusa**. Ea **nu** poate fi instantiata decat in aceasta functie.

Exemplu de clasa interna declarata intr-un **bloc oarecare:**

```
interface Hidden {
    public int value ();
}

class Outer {
    public Hidden getInnerInstance(int i) {
        if (i == 11) {

            class BlockInner implements Hidden {
                private int i = 11;

                public int value() {
                    return i;
                }
            }

            return new BlockInner();
        }

        return null;
    }
}
```

In acest exemplu, am definit clasa interna `BlockInner` in cadrul unui bloc `if`. **Atentie** insa: chiar daca declaratia clasei este in cadrul unui bloc `if`, **nu** inseamna ca declaratia va fi luata in considerare doar la rulare, in cazul in care conditia este adevarata. Semnificatia declararii clasei in acest bloc este legata strict de **vizibilitatea** acesteia. La compilare clasa va fi creata **indiferent** care este valoarea de adevar a conditiei `if`.

Trebuie mentionat faptul ca, in cadrul unei clase interioare unei metode, **nu** se poate face referire la variabilele locale **nefinale**. Iata un exemplu:

```
public void f() {
    final Student s = new Student();

    class AlterStudent {
        public void alterStudent() {
            s.name = ...        // OK
            s = new Student(); // GRESIT!
        }
    }
}
```

Acest lucru este justificat de urmatoarea observatie: variabilele locale unei functii se aloca in zona de memorie numita ***stiva***, urmand ca, la incheierea executiei metodei, locatia respectiva sa fie eliberata (eventual populata ulterior cu alt continut). O instanta a clasei interne poate exista si **dupa** incheierea executiei metodei (si poate deci executa instructiuni). Atasarea modificatorului `final` garanteaza faptul ca metoda `alterStudent` **nu** mai pot modifica, prin atribuire, locatia de pe stiva, **dupa** incheierea executiei metodei `f`, lucru ce ar putea conduce la **coruperea** informatiei.

Clase anonime

Exista multe situatii in care o clasa interna este instantiata intr-un singur loc (si este folosita prin *upcasting* la o clasa de baza sau interfata, ca in exemplele anterioare). In aceste situatii, numele clasei interne create este neimportant - el se va *pierde oricum pe drum*.

In Java putem crea **clase interne anonime** (fara nume). Exemplu:

```
interface Hidden {
    public int value();
}
```

```
class Outer {
    public Hidden getInnerInstance(int i) {
        return new Hidden() {
            private int i = 11;

            public int value() {
                return i;
            }
        };
    }
}

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

        Hidden in3 = out.getInnerInstance(11);
        System.out.println(in3.value());
    }
}
```

Observati modalitatea de declarare a clasei anonime. Sintaxa `return new Hidden() { ... }` spune urmatoarele lucruri:

- dorim sa intoarcem un obiect de tip `Hidden`
- acest obiect este instantiat imediat dupa `return`, folosind `new` (referinta intoarsa de `new` va fi *upcast* la clasa de baza: `Hidden`)
- numele clasei instantiate este absent (ea este anonima), insa ea este de *tipul* `Hidden` (prin urmare, va implementa metoda/metodele din interfata). Corpul clasei urmeaza imediat instantierii.

Constructia `return new Hidden() { ... }` este echivalenta cu a spune: *creeaza un obiect al unei clase anonime ce implementeaza `Hidden`*.

Clasele anonime **nu** pot avea **constructori** din cauza ca nu au nume (nu am sti cum sa numim constructorii). Ele pot fi initializate cu initializatori pentru campuri sau secvente de initializare. Aceasta restrictie asupra claselor anonime ridica o problema: in mod implicit, clasa de baza este creata cu constructorul default (ca in exemplu anterior). Ce se intampla daca dorim sa invocam un **alt constructor** al clasei de baza? In clasele normale acest lucru era posibil prin apelarea explicita, in prima linie din constructor a constructorului clasei de baza cu parametrii doriti, folosind `super`. In clasele interne acest lucru se obtine prin transmiterea parametrilor catre constructorul clasei de baza **direct** la crearea obiectului de tip clasa anonima:

```
new Student("Andrei") {
    ...
}
```

In acest exemplu, am instantiat o clasa anonima, ce extinde clasa `Student`, apeland constructorul acestei clase de baza cu parametrul `"Andrei"`.

Legatura cu clasa exterioara

Clase interne nestatice

In exemplele de mai sus, o instanta a **clasei interne** exista **doar in contextul** unei instante a **clasei exterioare**, astfel ca **poate accesa membrii** obiectului exterior direct.

Exemplu

```
interface Hidden {
    public int value ();
}

class Outer {
    public int outerMember = 9;

    class Inner implements Hidden {
        private int i = 1;

        public int value() {
            return i + outerMember;
        }
    }
}
```

In exemplul de mai sus, clasa interna `Inner` foloseste membrul `outerMember` al clasei exterioare. Asemanator, putem accesa **referinta la clasa externa** (in cazul nostru `Outer`), in felul urmator:

```
Outer.this;
```

Clase interne statice

Daca **nu** dorim o **legatura** intre obiectul clasei interne si obiectul clasei exterioare, atunci putem defini clasa noastra interna ca fiind **statica**. Pentru a intelege diferenta dintre clasele interne statice si cele nestatice (implicite) trebuie sa retinem urmatorul aspect: **clasele nestatice tin legatura cu obiectul exterior** in vreme ce **clasele statice nu pastreaza aceasta legatura**.

O clasa interna **statica** insemna:

- nu avem nevoie de un obiect al clasei externe pentru a crea un obiect al clasei interne
- nu putem accesa campuri nestatice ale clasei externe din clasa interna

Clasele interne nestatice nu pot avea campuri statice sau alte clase interne statice! In comparatie, clasele interne statice pot avea toate cele enumerate mai sus.

Exemplu de folosire a claselor statice:

```
class Outer {

    public int outerMember = 9;

    class NonStaticInner {

        private int i = 1;

        public int value() {
            return i + Outer.this.outerMember; // OK, putem accesa un membru al clasei exterioare
        }
    }

    static class StaticInner {

        public int k = 99;

        public int value() {
            k += outerMember; // EROARE, nu putem accesa un membru nestatic al clasei exterioare
            return k;
        }
    }
}

public class Test {

    public static void main(String[] args) {
        Outer out                   = new Outer ();

        Outer.NonStaticInner nonSt = out.new NonStaticInner(); // instantiere CORECTA pt o clasa nestatica
        Outer.StaticInner st       = out.new StaticInner();    // instantiere INCORECTA a clasei statice
        Outer.StaticInner st2      = new Outer.StaticInner();  // instantiere CORECTA a clasei statice
    }
}
```

Se observa ca folosirea membrului nestatic `outerMember` in clasa statica `StaticInner` este incorecta. De asemenea, se observa modalitatile **diferite** de instantiere a celor doua tipuri de clase interne (statice si nestatice):

- Folosim o **instanta** clasei exterioare - `out` (ca si in exemplele anterioare) pentru a instantia o clasa **nestatica**.
- Folosim **numele** claselor pentru a instantia o clasa **statica**. Folosirea lui `out` este incorecta.

Mostenirea claselor interne

Deoarece constructorul clasei interne trebuie sa se **ataseze** de un obiect al clasei exterioare, mostenirea unei clase interne este putin mai complicata decat cea obisnuita. Problema rezida in nevoia de a initializa legatura (ascunsa) cu clasa exterioara, in contextul in care in clasa derivata nu mai exista un *obiect default* pentru acest lucru (care era `NumeClasaExterna.this`).

```
class WithInner {
    class Inner {
        public void method() {
            System.out.println("I am Inner's method");
        }
    }
}
```

```
class InheritInner extends WithInner.Inner {

    InheritInner() {}           // EROARE, avem nevoie de o legatura la obiectul clasei exterioare
    InheritInner(WithInner wi) { // OK
        wi.super();
    }
}

public class Test {
    public static void main(String[] args) {
        WithInner wi    = new WithInner();
        InheritInner ii = new InheritInner(wi);
        ii.method();
    }
}
```

Observam ca `InheritInner` mosteneste doar `WithInner.Inner` insa sunt necesare:

- **parametrul constructorului** `InheritInner`, de tip clasa externa (`WithInner`)
- linia din constructorul `InheritInner`: `wi.super()`.

Utilizarea claselor interne

Clasele interne pot parea un mecanism greoi si uneori artificial. Ele sunt insa foarte utile in urmatoarele situatii:

- Rezolvam o problema complicata si dorim sa cream o clasa care ne ajuta la dezvoltarea solutiei dar:
  - **nu** dorim sa fie **accesibila** din exterior sau
  - **nu** mai are **utilitate** in alte zone ale programului
- Implementam o anumita interfata si dorim sa intoarcem o referinta la acea interfata, **ascunzand** in acelasi timp implementarea.
- Dorim sa folosim/extindem functionalitati ale mai **multor** clase, insa in JAVA nu putem extinde decat o singura clasa. Putem defini insa clase interioare. Acestea pot **mosteni** orice clasa si au, in plus, acces la obiectul clasei **exterioare**.
- Implementarea unei **arhitecturi de control**: O arhitectura de control este o *arhitectura-aplicatie* marcata de nevoia de a trata evenimente intr-un sistem bazat pe evenimente. Unul din cele mai importante sisteme de acest tip este GUI (graphical user interface). Biblioteca `Java Swing` este o arhitectura de control care rezolva foarte elegant problema GUI si foloseste intens clase interne (vezi exercitiul 3).

Exercitii

1. (**3p**) Extindeti clasa `MyArrayList` (pe care o gasiti in solutiile laboratorului 3) cu un iterator. In esenta, un iterator este un obiect care permite parcurgerea unei liste.
   - Scrieti o interfata `Iterator`, ce contine metodele:
     - `hasNext()`: intoarce `true` daca mai exista elemente neparcurse
     - `next()`: intoarce elementul urmator, si avanseaza pe pozitia urmatoare din lista
       - daca nu mai exista elemente neparcurse, intoarce `0`
       - daca lista a fost modificata dupa crearea iteratorului, intoarce `-1` (iteratorul afirma ca nu mai poate garanta parcurgerea corecta a listei).
   - Definiti in clasa `MyArrayList` o metoda, `iterator()`, care intoarce o implementare a interfetei `Iterator`, descrisa mai sus, ce va permite parcurgerea, element cu element, a listei. Implementarea trebuie realizata folosind **clase interne neanonime**. Care este alegerea potrivita: clase **statice** sau **nestatice**?
   - Realizati o parcurgere a unei liste utilizand **exclusiv** un iterator.
2. (**3p**) Implementati un sistem de criptare/decriptare a unui sir de caractere.
   - Sistemul foloseste doua mecanisme de criptare. Primul codifica un caracter, avand codul ASCII `x`, prin caracterul cu codul ASCII `x + 1`. Al doilea sistem este similar, insa caracterul codificat are codul ASCII `x - 5`.
   - Pentru a defini comportamentul specific criptarii, scrieti o interfata `Encrypter`, ce contine doua metode:
     - `encrypt`
     - `decrypt`
   - Definiti o clasa `EncrypterFactory`. Aceasta va contine o metoda `get` care intoarce o instanta avand tipul `Encrypter`, reprezentand unul din cele doua mecanisme de criptare, ales la intamplare. Ambele implementari trebuie realizate folosind **clase interne neanonime**. Care este alegerea fireasca: clase **statice** sau **nestatice**?
   - Testati

     Atentie: clasa `String` este *immutable* i.e. nu permite realizarea de modificari asupra instantelor sale. Pentru a accesa si modifica direct caracterele, convertiti, mai intai, obiectul `String` la un vector de `char`, folosind metoda `String.toCharArray()`.
3. (**6p**) Imbogatiti clasa `MyArrayList` cu un mecanism de notificare pentru operatiile de adaugare si stergere din lista. Lista va dobandi rolul de *observat* (*observed*), si vom defini entitati suplimentare, cu rolul de *observator* (*observer*). Observatorii se vor

inregistra la lista, iar aceasta din urma, la infaptuirea unei operatii de adaugare sau stergere, va semnaliza tuturor observatorilor inregistrati aparitia evenimentului corespunzator. Un astfel de eveniment va contine mai multe informatii.

- Definiti interfata `ListObserver` ce desemneaza un observator pe o lista, cu metodele:
  - `elementAdded(ListEvent e)`: apelata pe un observator cand un element este adaugat in lista observata de acesta
  - `elementRemoved(ListEvent e)`: apelata pe un observator cand un element este inlaturat din lista observata de acesta
- Definiti interfata `Observed` ce desemneaza o entitate observata, cu metoda:
  - `registerObserver(ListObserver o)`: apelata pe un obiect observat cand se doreste inregistrarea unui observator de lista
- Definiti interfata `ListEvent`, cu metodele:
  - `getList()`: intoarce lista observata, care a generat evenimentul de adaugare sau stergere
  - `getElement()`: intoarce elementul implicat in eveniment
  - `getDuration()`: intoarce durata evenimentului, in nanoscunde (vezi `System.nanoTime()`).
- Modificati clasa `MyArrayList`, astfel incat sa implementeze interfata `Observed`, si sa-si asume rolul de lista observata. Acest lucru presupune doua etape:
  - expunerea mecanismului de **inregistrare** de noi observatori
  - **generarea** de evenimente la adaugare si stergere. Folositi **clase anonime**.
- Instantiati clasa `MyArrayList`, si inregistrati doi observatori, folosind **clase anonime**:
  - Primul va afisa pe ecran informatiile obtinute din eveniment.
  - Al doilea va popula o a doua instanta `MyArrayList` cu elementele pentru care durata evenimentului de **eliminare** depaseste un anumit prag, exprimat in milisecunde. Gasiti o modalitate de a accesa lista din **exteriorul** clasei observator.

Nota: Acest mecanism de **decuplare** a obiectelor observabile, ce pot genera evenimente, de obiectele observator, ce receptioneaza evenimentele respective, poarte numele de ***Observer Pattern***.

- **Solutii**

Adus de la "http://cursuri.cs.pub.ro/~poo/wiki/index.php/Clase_interne"

- Ultima modificare 08:21, 24 noiembrie 2012.
- Această pagină a fost vizitată de 8.403 ori.
-
- Politica de confidenţialitate
- Despre POO
- Termeni