

Clase interne

- Responsabil: Mihai Burduselu
- Data publicării: 10.11.2013
- Data ultimei modificări: 08.10.2017

Obiective

Scopul acestui laborator este prezentarea conceptului de clasă internă și modalitățile de creare și folosire a claselor interne în Java.

Aspectele urmărite sunt:

- prezentarea tipurilor de clase interne
- diferențele dintre clase interne statice și cele ne-statice
- utilitatea claselor interne

Introducere

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

Clasele interne sunt de mai multe tipuri, în funcție de modul de a le instanția și de relația lor cu clasa exterioră:

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

Unul din avantajele claselor interne este comportamentul acestora ca un **membru** al clasei. Asta face ca o clasă internă să poată avea acces la toți membrii clasei de care aparține (*outer class*), inclusiv cei *private*. În plus, aceasta poate avea modificatorii permisi metodelor și variabilelor claselor. Astfel, o clasă internă poate fi nu numai *public*, *final*, *abstract* dar și *private*, *protected* și *static*.

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 `Outer.class` și `Outer$Inner.class`, însă execuția fișierului `Outer$Inner.class` nu este permisă.

Test.java

```
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());
    }
}
```

În exemplul de mai sus, o dată ce avem o instanță a clasei Outer, sunt folosite două modalități de a obține o instanță a clasei Inner (definită în interiorul clasei Outer):

- definim o metodă getInnerInstance, care creează și întoarce o astfel de instanță;
- instanțiem efectiv Inner; observați cu atenție sintaxa folosită! Pentru a instanția Inner, avem nevoie de o instanță Outer: `out.new Inner(10);`

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

```
Outer.this;
```

Atenție ! Deși dintr-o clasă internă putem accesa membrii clasei externe, nu îi putem modifica ! (Hint: Pass by value) Pentru o explicație detaliată citiți [Link1](#) și [Link2](#) .

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**.

[Test.java](#)

```
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
        // 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());
    }
}
```

Observați definirea interfeței Hidden. Ea este utilă pentru a putea **asocia** clasei HiddenInner un tip, care să ne permită folosirea instanțelor acesteia, altfel tipul ei nu ar fi fost vizibil pentru că a fost declarată `private`. Observați, de asemenea, încercările eronate de a instanția HiddenInner. Cum clasa internă a fost declarată `private`, acest tip nu mai este vizibil în exteriorul clasei Outer.

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 complet și există modalități mai “obscure” de a le folosi: clasele interne pot fi definite și în cadrul metodelor sau al unor blocuri arbitrare de cod.

Clase interne în metode

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

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

[Test.java](#)

```
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();
        // EROR: clasa FuncInner nu este vizibila

        Hidden in3 = out.getInnerInstance();
    }
}
```

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

Clasele interne declarate în metode nu pot folosi variabilele declarate în metoda respectivă și nici parametrii metodei. Pentru a le putea accesa, variabilele trebuie declarate `final`, ca în exemplul următor. Această restricție se datorează faptului că variabilele și parametrii metodelor se află pe segmentul de stivă (zonă de memorie) creat pentru metoda respectivă, ceea ce face ca ele să nu fie existente 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.

```
public void f() {
    final Student s = new Student();
    // s trebuie declarat final ca sa poata fi accesat din AlterStudent

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

Clase interne în blocuri

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

```
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;
    }
}
```

```
}  
}
```

În acest exemplu, clasa internă `BlockInner` 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 adevărată.

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*.

Clase anonime

Exista multe situații în care o clasă internă este instanțiată într-un singur loc (și este folosită prin upcasting la o clasă de bază sau interfață), ceea ce face ca numele clasei să nu mai fie important, iar tipul ei poate fi un subtip al unei clase sau o implementare a unei interfețe. Singurele metode care pot fi apelate pe o clasă anonimă sunt cele ale tipului pe care îl extinde sau implementează.

În Java putem crea **clase interne anonime** (fără nume) ca în exemplul următor:

```
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());  
    }  
}
```

Observați modalitatea de declarare a clasei anonime. Sintaxa `return new Hidden() { ... }` reprezintă următoarele:

- dorim să întoarcem un obiect de tip `Hidden`

- acest obiect este instanțiat imediat după return, folosind new (referința întoarsă de new va fi upcast la clasa de baza: Hidden)
- numele clasei instanțiate este absent (ea este anonimă), însă ea este de tipul Hidden, prin urmare, va implementa metoda/metodele din interfață (cum e metoda value). Corpul clasei urmează imediat instanțierii.

Construcția `return new Hidden() { ... }` este echivalentă cu a spune: *crează un obiect al unei clase anonime ce implementează Hidden*.

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.

Constructori

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*.

Ce se întâmplă dacă dorim să invocăm un **alt constructor** al clasei de bază? În clasele normale acest lucru era posibil prin apelarea explicită, în prima linie din constructor a constructorului clasei de bază cu parametrii doriți, folosind `super`. În clasele interne acest lucru se obține prin transmiterea parametrilor către constructorul clasei de bază **direct** la crearea obiectului de tip clasă anonimă:

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

În acest exemplu, am instanțiat o clasă anonimă, ce extinde clasa `Student`, apelând constructorul clasei de bază cu parametrul `"Andrei"`.

Clase interne statice

În secțiunile precedente, s-a discutat doar despre clase interne a caror instanțe există doar în contextul unei instanțe a clasei exterioare, astfel că poate accesa membrii obiectului exterior direct. De asemenea, am menționat că fiind membri ai claselor exterioare, clasele interne pot avea modificatorii disponibili pentru metode și variabile, dintre care și `static` (clasele exterioare nu pot fi statice!). 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 a înțelege diferența dintre clasele interne statice și cele nestatice trebuie să reținem următorul aspect: **clasele nestatice țin legătura cu obiectul exterior** în vreme ce **clasele statice nu păstrează această legătură**.

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)

Test.java

```
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
    }
}
```

În exemplul de mai sus se observă că folosirea membrului nestatic `outerMember` în clasa statică `StaticInner` este incorectă. De asemenea, se observă modalitățile diferite de instanțiere a celor două tipuri de clase interne (statice și nestatice):

- folosim o instanță a clasei exterioare - `out` (ca și în exemplele anterioare) pentru a instanția o clasă nestatică.
- folosim numele claselor pentru a instanția o clasă statică. Folosirea lui `out` este incorectă.

- *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 A.B, când o facem statică?*
 - în interiorul clasei B nu avem nevoie de nimic specific instanței clasei externe A, deci nu avem nevoie de o instanță a acesteia → o facem statică

Terminologia *nested classes* vs *inner classes*:

Clasele interne normale, cele anonime și cele interne blocurilor și metodelor sunt *inner classes* datorită relației pe care o au cu clasa exterioară (depind de o instanță a acesteia). Termenul de *nested classes* se referă la definirea unei clase în interiorul altei clase, și cuprinde atât *inner classes* cât și clasele statice interne. De aceea, claselor statice interne li se spune *static nested classes* și nu *static inner classes*.

Moștenirea claselor interne

Deoarece constructorul clasei interne trebuie să 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 (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();
    }
}
```

Observăm ca `InheritInner` moșteneste doar `WithInner.Inner` însă sunt necesare:

- parametrul constructorului `InheritInner` trebuie să fie de tipul clasei externă (`WithInner`)
- linia din constructorul `InheritInner`: `wi.super()`.

Utilizarea claselor interne

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

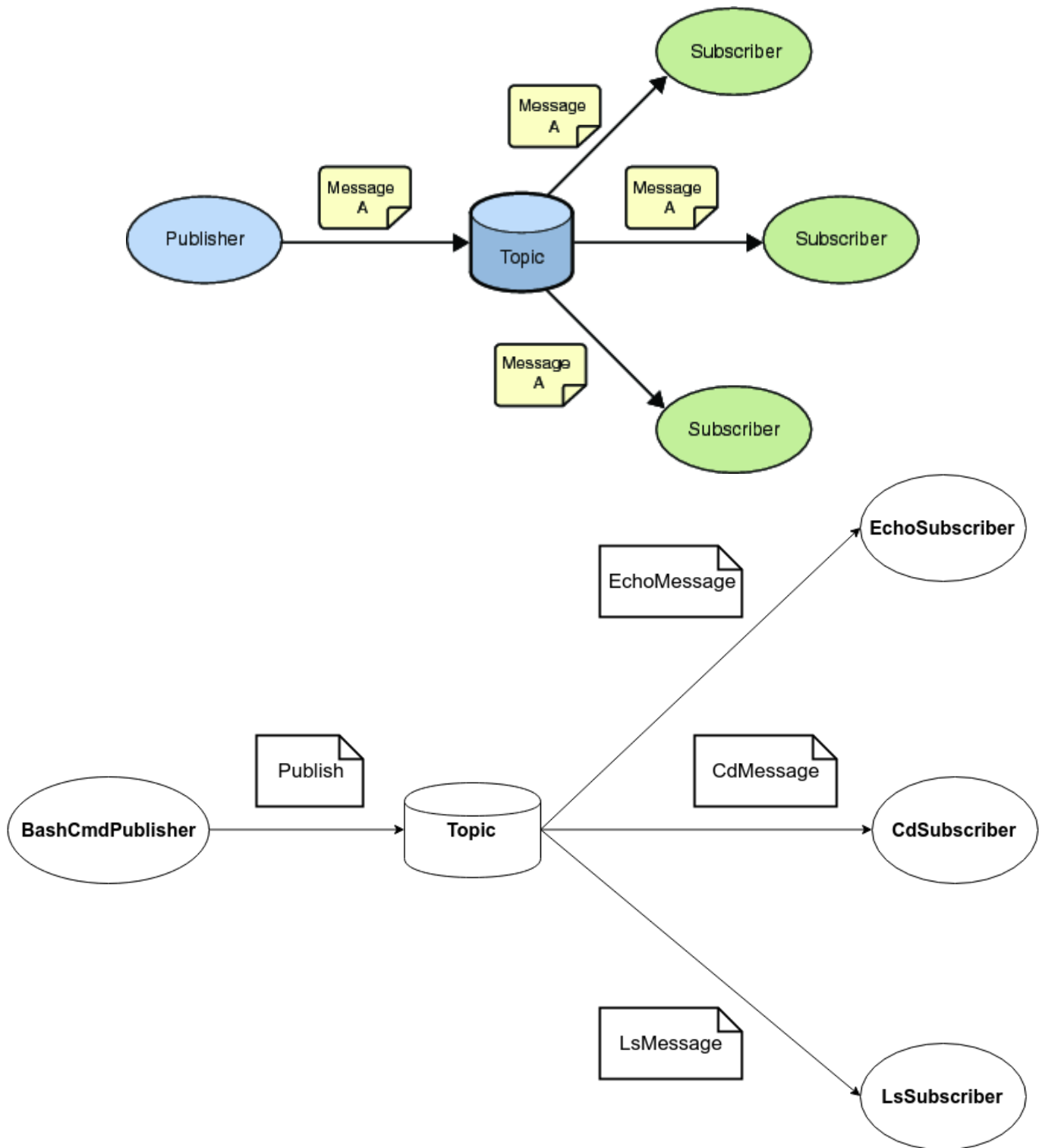
- Rezolvăm o problemă complicată și dorim să creăm o clasă care ne ajută la dezvoltarea soluției dar:
 - **nu** dorim să fie **accesibilă** din exterior sau
 - **nu** mai are **utilitate** în alte zone ale programului
- Implementăm o anumită interfață și dorim să întoarcem o referință la acea interfață, **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](#), [AWT](#), [SWT](#) sunt arhitecturi de control care folosesc intens clase interne. De exemplu, în Swing, pentru **evenimente** 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

Implementați un terminal bash simplu pornind de la scheletul de cod. Comenzile pe care va știți să le executați sunt: **echo**, **cd**, **ls** și **history**. Bash-ul va citi comenzi de la tastatură până când va primi comanda **exit** când se va închide (programul se termină).

- În clasa **BashUtils** din pachetul **bash** vom implementa fiecare comandă ca o clasă internă.
- În clasa **Bash** din pachetul **bash** vom citi comenzi de la tastatură și le vom trimite spre **BashUtils** spre a fi executate.
- Mecanismul de funcționare va fi de tipul [Publisher-Subscriber](#).
 - Concret: Bash-ul va citi comenzile de la tastatură și le va **publica** către toți subscriberii săi. Bash-ul va funcționa ca un **Publisher**.
 - Utilitățile, **ls**, **echo**, **cd** și **history**, care știu cum să execute comenzile se vor înregistra la **Publisher** folosind metoda **subscribe** a acestuia și vor fi anunțate când o nouă comandă este primită. Ele vor funcționa ca **Subscriberi**.



- În scheletul de cod veți găsi 2 interfețe:
 - **CommandPublisher** având metodele:

```

/**
 * Subscribe a CommandSubscriber to this Publisher so that it is
 * notified when a command is received.
 */
void subscribe(CommandSubscriber subscriber);

/**
 * Publish a Command to all registered subscribers.
 * Usually by calling the method defined in the CommandSubscriber

```

```
interface.
    */
    void publish(Command command);
```

- **CommandSubscriber** având metoda:

```
/**
 * Executes the given command.
 * We can call this method from a Publisher to tell the Subscriber that
 * a command was received.
 */
void executeCommand(Command c);
```

- Observăm că un **CommandPublisher** face **publish** la un obiect de tip **Command**, iar un **CommandSubscriber** primește în metoda **executeCommand** un astfel de obiect ca parametru.
 - Găsiți în scheletul de cod implementarea clasei **Command**. Acesta este obiectul prin care Publisherul și Subscriberii comunică aka își trimit date.

[TODO 1] (1.5p) Din clasa **Bash** vom **publica** comenzile prin interfața **CommandPublisher**. În acest sens în clasa **Bash** vom crea o clasă internă **BashCommandPublisher** ce implementează interfața **CommandPublisher**.

- În această clasă creați o **listă** de elemente de tip **CommandSubscriber**.
- Apoi va trebui să implementați metodele:
 - **subscribe**-prin care adăugăm un subscriber în lista de subscrieri
 - **publish**-în care iterăm prin lista de subscrieri și trimitem evenimentul către subscrieri apelând metoda definită în interfața **CommandSubscriber** (în cazul de față **executeCommand**)

[TODO 2] (1p) Un obiect de tipul **Bash** va avea:

- un director current reținut în membrul **currentDirectory** care este de tipul **Path**
- un istoric al comenzilor care este de tipul **StringBuffer** .

De ce este mai util să folosim un **StringBuffer** și nu un **String** simplu?

String vs StringBuffer ?

- În constructorul clasei **Bash** vom inițializa **history** și apoi **currentDirectory** cu calea către directorul curent **“.”**. Hint: [Paths.get](#)
- Instantiați și obiectul de tip **CommandPublisher** definit la exercițiul anterior. Prin intermediul lui vom publica comenzi din **Bash** în sistem.

[TODO 3] (2p) În metoda **start** din **Bash** vom citi de la tastatură comenzi pe câte o linie folosind **Scanner**.

- Când se citește string-ul **exit** programul se termină.
- Pentru orice altă comandă vom crea un nou **Java Thread** de pe care vom **publica** comanda către subscrieri prin instanța de **CommandPublisher** creată la exercițiul 1.

- Creați și instanțiați o clasă internă anonimă ce **extinde** clasa **Thread**. Clasa va trebui să implementeze metoda **run** prin care îi spunem thread-ului ce să facă (în cazul nostru să apeleze metoda **publish**).

Pentru a porni un Thread apelăm metoda **start** a acestuia. Implementarea metodei **run** și instanțierea Thread-ului nu îl lansează în execuție.

```
Thread t = new Thread() {  
    public void run() {  
        // Do some work  
    }  
};  
t.start();
```

[TODO 4] (1p) Implementați comanda **echo** ca o clasă internă în **BashUtils**.

- Clasa va trebui să implementeze interfața **CommandSubscriber**.
- Metoda **executeCommand** va trebui să:
 - verifice dacă comanda primită începe cu "echo". Altfel nu va trebui să facă nimic.
 - să afișeze la consolă șirul aflat după cuvântul cheie "echo"

```
>echo bla  
bla
```

- Creați un obiect de tipul clasei **Echo** în constructorul clasei **Bash**. Înregistrați obiectul ca **subscriber** la instanța de **CommandPublisher** folosind metoda **subscribe**.
- Testați rulând metoda **main** din clasa **LinuxOS**.

[TODO 5] (1.5p) Implementați comanda **cd** care schimbă directorul curent.

- Clasa va trebui să implementeze interfața **CommandSubscriber**.
- Metoda **executeCommand** va trebui să:
 - verifice dacă comanda primită începe cu "cd".
 - să schimbe variabila **currentDirectory** cu noua cale. Funcția **cd** va face **append** la calea deja existentă în **currentDirectory**. Hint: [Paths.get](#)

```
currentDirectory = E:FacultateLab05.  
>cd .idea  
currentDirectory = E:FacultateLab05..idea
```

- Creați un obiect de tipul clasei **Cd** în constructorul clasei **Bash**. Înregistrați obiectul ca **subscriber** la instanța de **CommandPublisher** folosind metoda **subscribe**.
- Testați rulând metoda **main** din clasa **LinuxOS**.

[TODO 6] (2p) Implementați comanda **ls** care afișează conținutul directorului curent **currentDirectory**

- Clasa va trebui să implementeze interfața **CommandSubscriber**.

- Metoda **executeCommand** va trebui să:
 - verifice dacă comanda primită este "ls" fără parametrii.
 - să itereze prin conținutul directorului curent și să afișeze numele fișierelor la consolă, pe câte o linie fiecare. Hint: [cum citim conținutul unui director în Java?](#)

```
>ls
E:FacultateLab05.
.idea
clase-interne2.iml
out
src
```

- **currentDirectory** are tipul **Path**. Putem obține un obiect de tip **File** astfel:

```
File folder = currentDirectory.toFile();
```

- Creați un obiect de tipul clasei **Ls** în constructorul clasei **Bash**. Înregistrați obiectul ca **subscriber** la instanța de **CommandPublisher** folosind metoda **subscribe**.
- Testați rulând metoda **main** din clasa **LinuxOS**.

[TODO 7] (1p) Implementați comanda **history** care afișează la consolă conținutul membrului **StringBuffer history** din **Bash**.

```
>history
History is: ls | | cd .idea | ls | history |
```

- Clasa va trebui să implementeze interfața **CommandSubscriber**.
- Metoda **executeCommand** va trebui să:
 - să adauge fiecare comanda primită în StringBuffer-ul **history** Hint: [StringBuffer append](#)
- dacă comandă primită este "history", să afișeze la consolă conținutul lui **history**
- Creați un obiect de tipul clasei **Echo** în constructorul clasei **Bash**. Înregistrați obiectul ca **subscriber** la instanța de **CommandPublisher** folosind metoda **subscribe**.
- Testați rulând metoda **main** din clasa **LinuxOS**.

Resurse

- [Schelet](#)
- [PDF laborator](#)

Referințe

1. Kathy Sierra, Bert Bates. *SCJP Sun Certified Programmer for Java™ 6 - Study Guide*. Chapter 8 - Inner Classes ([available online](#))

From:

<http://elf.cs.pub.ro/poo/> - **Programare Orientată pe Obiecte**

Permanent link:

<http://elf.cs.pub.ro/poo/laboratoare/clase-interne>

Last update: **2017/10/08 09:04**

