Clase interne

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 anonime (*anonymous inner classes*)
- clase interne statice (*static nested 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 clasa internă sa poata avea acces la toți membrii clasei de care aparține (*outer class*), inclusiv cei `private`. În plus, aceasta poate avea modificatorii permiși metodelor și variabilelor claselor. Astfel, o clasa 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 Car {
    class Engine {
```

```
        private int fuelCapacity;

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

        public int getFuelCapacity() {
            return fuelCapacity;
        }
    }

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

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

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

        System.out.println(firstEngine.value());
        System.out.println(secondEngine.value());
    }
}
```

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

- definim o metodă `getEngine`, care creează și întoarce o astfel de instanță;
- instanțiem efectiv `Engine`; observați cu atentie sintaxa folosita! Pentru a instanția `Engine`, avem nevoie de o instanta `Car`: `car.new Engine(10);`

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**.

Test.java

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

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

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

        public int getFuelCapacity() {
            return fuelCapacity;
        }
    }

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

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

        Car.FerrariEngine firstEngine = car.getEngine();  // va genera
eroare, deoarece
                                                          // tipul
Car.FerrariEngine nu este vizibil
        Car.FerrariEngine secondEngine = new Car().new
FerrariEngine(10); // din nou eroare

        Engine thirdEngine = car.getEngine();           // acces corect
la o instanta FerrariEngine
        System.out.println(thirdEngine.getFuelCapacity());
    }
}
```

Observați definirea interfeței `Engine`. Ea este utilă pentru a putea **asocia** clasei `FerrariEngine` un `tip`, care să ne permita folosirea instanțelor acesteia, altfel tipul ei nu ar fi fost vizibil pentru ca a fost declarată `private`. Observați, de asemenea, încercarile eronate de a instanția `FerrariEngine`. Cum clasa internă a fost declarată `private`, acest tip nu mai este vizibil in exteriorul clasei `Car`.

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 clasa 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 Engine {
    public int getFuelCapacity();
}

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

            public int getFuelCapacity() {
                return fuelCapacity;
            }
        };
    }
}

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

        Engine engine = car.getEngine(11);
        System.out.println(engine.getFuelCapacity());
    }
}
```

IntelliJ sugerează înlocuirea cu funcții lambda însă acest concept nu este acoperit în laborator. Pentru detalii suplimentare urmăriți acest exemplu

Observați modalitatea de declarare a clasei anonime. Sintaxa `return new Engine() { … }` reprezintă urmatoarele:

- dorim sa întoarcem un obiect de tip `Engine`
- acest obiect este instanțiat imediat dupa `return`, folosind `new` (referința întoarsă de `new` va fi `upcast` la clasa de baza: `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 `value`). Corpul clasei urmeaza imediat instanțierii.

Construcția `return new Engine() { … }` este echivalentă cu a spune: *creează un obiect al unei clase anonime ce implementeaza `Engine`*.

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 știi 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 clasa 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 ale 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 aceteia, 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 Student {
    public int grade = 9;

    class Gradebook {
        private int i = 1;
```

```
        public int updateGrade() {
            return i + Student.this.grade; // OK, putem accesa un
membru al clasei exterioare
        }
    }

    static class EvilGradebook {
        public int k = 99;

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

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

        Student.Gradebook out = student.new Gradebook(); // instanțiere
CORECTĂ pentru o clasă nestatică
        Student.EvilGradebook badStudent = student.new EvilGradebook();
// instanțiere INCORECTĂ a clasei statice
        Student.EvilGradebook smartStudent = new
Student.EvilGradebook();     // instanțiere CORECTĂ a clasei statice
    }
}
```

În exemplul de mai sus se observă că folosirea membrului nestatic `grade` în clasa statică `EvilGradebook` 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 - `student` (ca și în exemplele anterioare) pentru a instanția o clasă nestatică.
- folosim numele claselor pentru a instanția o clasă statică. Folosirea lui `student` 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 si cele interne blocurilor si 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*.

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.

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 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).

Test.java

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

class Car {
    public Engine getEngine() {
        class FerrariEngine implements Engine {
            private int fuelCapacity = 11;

            public int getFuelCapacity() {
                return fuelCapacity;
            }
        }

        return new FerrariEngine();
    }
}

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

        Car.FerrariEngine badEngine = car.getEngine();
        // EROARE: clasa FerrariEngine nu este vizibila

        Engine goodEngine = car.getEngine();
```

```
        System.out.println(goodEngine.getFuelCapacity());
    }
}
```

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 si parametrii metodelor se află pe segmentul de stivă (zonă de 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. Pentru o explicație detaliată citiți Link1 și Link2.

```
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 = "Andrei"           // OK
            s = new Student();          // GRESIT!
        }
    }
}
```

Clase interne în blocuri

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

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

class Car {
    public Engine getEngine(int fuelCapacity) {
        if (fuelCapacity == 11) {
            class FerrariEngine implements Engine {
                private int fuelCapacity = 11;

                public int getFuelCapacity() {
                    return fuelCapacity;
                }
            }

            return new FerrariEngine();
        }
```

```
        return null;
    }
}
```

În acest exemplu, clasa internă `FerrariEngine` 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 FerrariEngine extends Car.Engine {
    FerrariEngine() {
    } // EROARE, avem nevoie de o legatura la obiectul clasei exterioare

    FerrariEngine(Car car) { // OK
        car.super();
    }
}

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

Observăm ca `FerrariEngine` moșteneste doar `Car.Engine` însa sunt necesare:

- parametrul constructorului `FerrariEngine` trebuie sa fie de tipul clasei externă (`Car`)
- linia din constructorul `FerrariEngine`: `car.super()`.

Utilizarea claselor interne

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

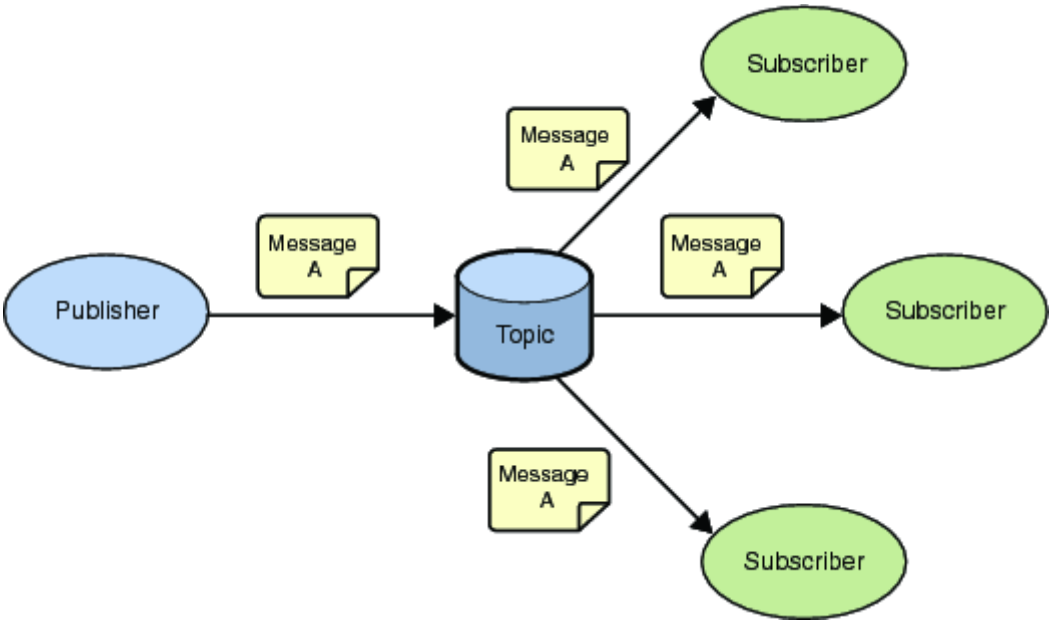
- Rezolvăm o problemă complicată și dorim să creăm o clasă care ne ajută la dezvoltarea soluției dar:
  - **nu** dorim să fie **accesibilă** din exterior sau
  - **nu** mai are **utilitate** în alte zone ale programului
- Implementăm o anumită interfață și dorim să întoarcem o referință la acea interfață, **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, 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++;
    }
    });
```

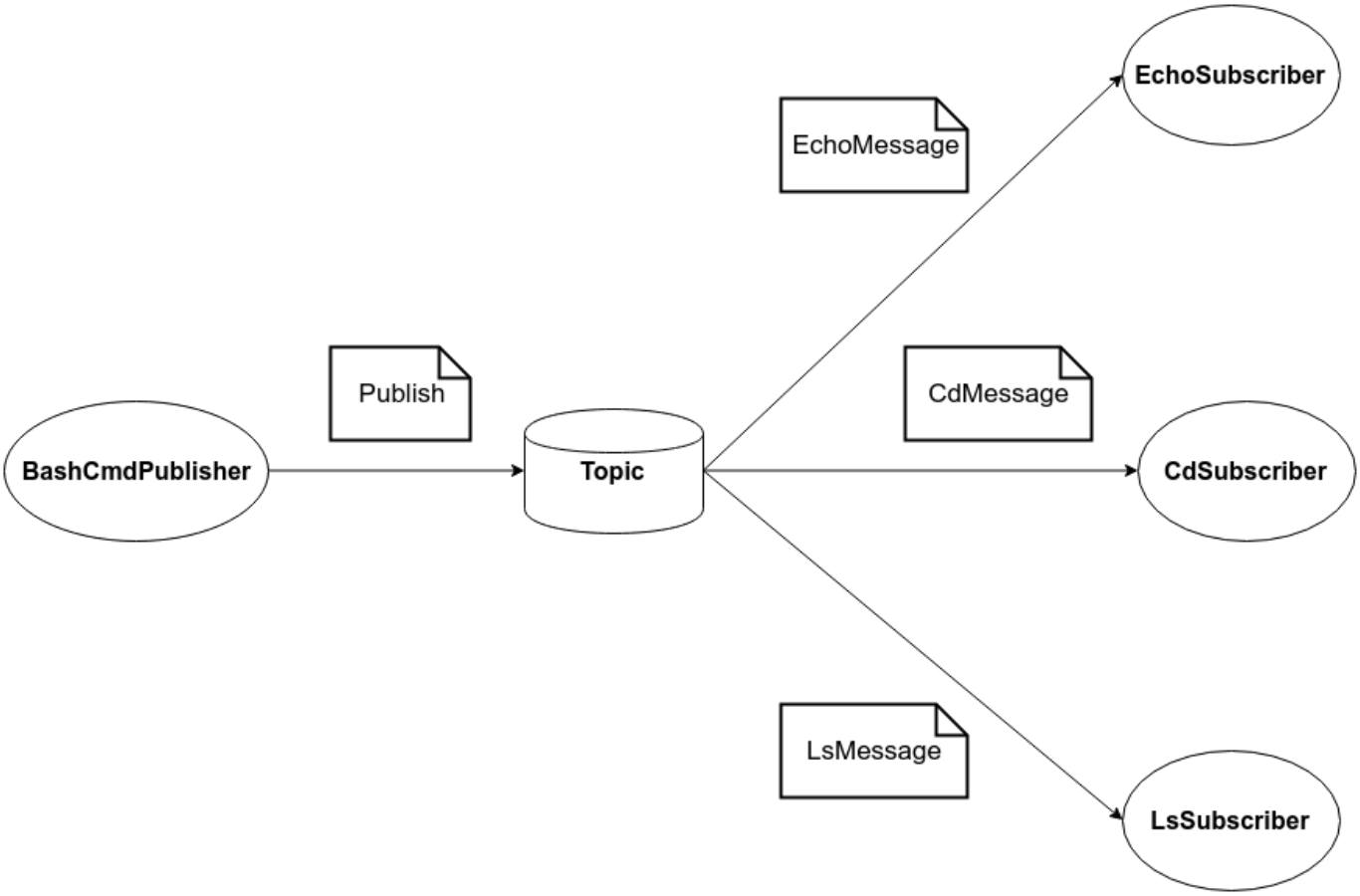
Exerciții

Implementați un terminal bash simplu pornind de la scheletul de cod. Comenzile pe care va știi să le execute 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**.
- Utilitarele, **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(CommandSubscriber)
    - publish(Command)

    - **CommandSubscriber** având metoda:
        - executeCommand(Command
- 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.

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 iteram prin lista de subscriberi și trimitem evenimentul către subscriberi apelând metoda definită în interfața **CommandSubscriber** (în cazul de față **executeCommand**)

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?
Sting vs StringBuffer ?

- În constructorul clasei **Bash** vom inițializa **history** și apoi **currentDirectory** cu calea către directorul curent ".". Hint: Paths.get
- Instantiati și obiectul de tip **CommandPublisher** definit la exercițiul anterior. Prin intermediul lui vom publica comenzi din **Bash** în sistem.

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 subscriberi prin instanța de **CommandPublisher** creată la exercițiul 1.
    - Creați și instantiati 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 instantierea Thread-ului nu îl lansează în execuție.

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

```
t.start();
```

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”
- 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**.

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
- 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**.

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?
  - **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**.

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 **History** în constructorul clasei **Bash**. Înregistrați obiectul ca **subscriber** la instanța de **CommandPublisher** folosind metoda **subscribe**.

- Testați rulând metoda **main** din clasa **LinuxOS**.

Resurse

- Schelet
- Soluție
- PDF laborator

Referințe

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