Input/Output

De la POO

Salt la: Navigare, căutare

Cuprins

[ascunde] [ascunde]

- 1 Introducere
  - 1.1 Streams
- 2 Stream-uri octet
  - 2.1 InputStream
  - 2.2 OutputStream
  - 2.3 Mod de utilizare
  - 2.4 Stream-urile standard
- 3 Stream-uri caracter
- 4 Stream-uri punte octet-caracter
- 5 Acces aleator
- 6 Manipularea fisierelor
- 7 Exercitii
- 8 Referinte

Introducere

Operatiile de intrare/iesire sunt realizate, in general, cu ajutorul claselor din **pachetul** `java.io`. Acest pachet contine un numar foarte mare de clase, intimidant la inceput, desi, paradoxal, design-ul claselor din `Java IO` previne "explozia" de clase. In platforma 1.0, biblioteca I/O era centrata in jurul operatiilor la nivel de **octet**. Variantele ulterioare includ clase pentru lucrul cu **caractere**. Este necesar sa intelegem cum a evoluat biblioteca `java.io`, pentru a intelege ce clase sunt folosite si in ce momente, de ce unele metode au devenit *deprecated* (se recomanda sa nu mai fie folosite) etc.

Streams

Bibliotecile I/O folosesc conceptul de **stream** (*flux*), ce reprezinta orice **sursa** sau **consumator** de date care este capabil sa produca sau sa primeasca **unitati** de date, intr-o maniera **secventiala**. Stream-ul **ascunde** detaliile a ceea ce se intampla cu datele in interiorul entitatii I/O, care poate fi:

- **fisier** de pe disc
- **program** (care poseda cele 3 stream-uri standard: `in`, `out`, `err`)
- etc

Clasele Java pentru I/O sunt separate din punct de vedere al operatiei: **input** si **output**. Exista doua linii mari de lucru cu stream-uri, si, anume, la nivel de:

- **octet**: clasele derivate din `InputStream` si `OutputStream`. Sunt asociate adesea cu fisierele **binare**.
- **caracter**: clasele derivate din `Reader` si `Writer`. Sunt asociate adesea cu fisierele **text**. Acest lucru nu inseamna ca un fisier text nu poate fi privit ca un flux **binar**, dar, daca informatia se doreste a fi interpretata sub forma sirurilor de caractere, este necesara perspectiva **caracter**.

Majoritatea functiilor din aceste clase pot arunca **exceptii** de tip **`IOException`** sau derivate din aceasta, cum este **`FileNotFoundException`**. Acestea sunt **checked**, deci vor trebui prinse.

Stream-uri octet

`InputStream`

Clasa **`InputStream`** se situeaza in **varful** ierarhiei de clase care descriu fluxuri octet **sursa**:

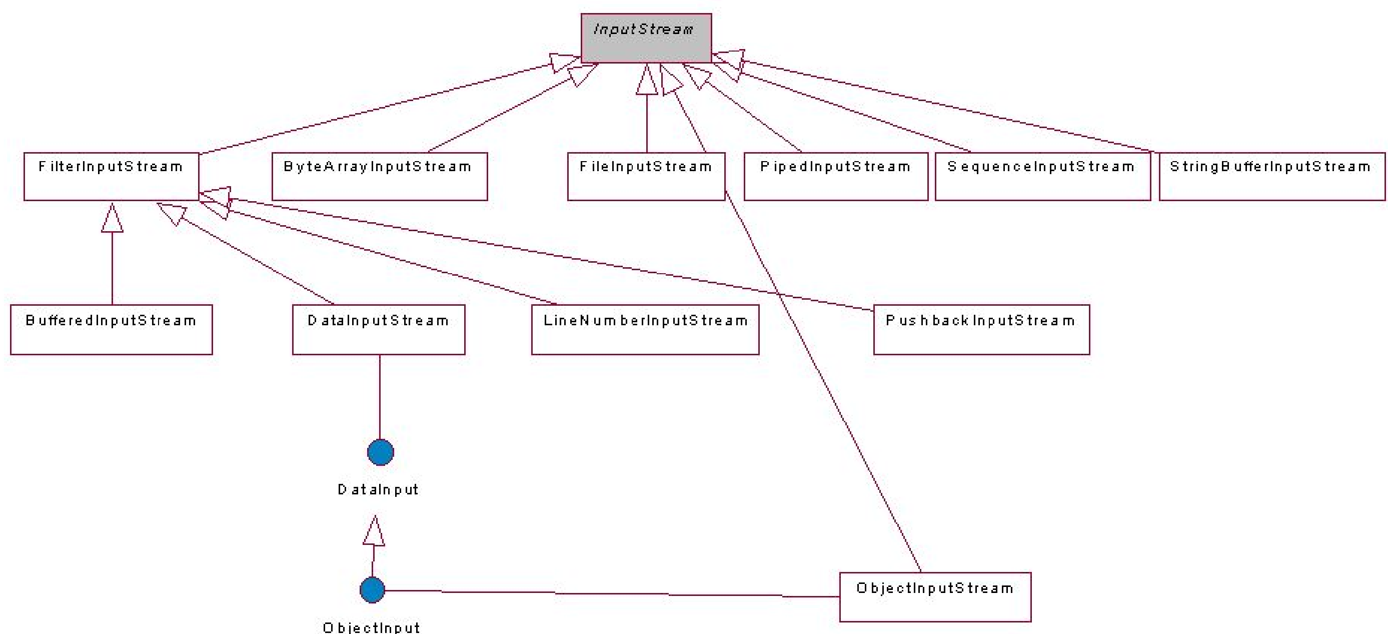
- **sir de octeti**
- obiect **`String`**
- **fisier**
- **pipe** (care functioneaza asemanator unei "tevi": datele sunt introduse pe o parte si scoase prin cealalta parte, in ordinea in care au fost introduse).
- **socket** (conexiune de retea)

Clasa este **abstracta** si ofera urmatoarea functionalitate:

- `abstract int read()` – citeste si intoarce un octet, iar, in caz de EOF (*end of stream*), se intoarce -1. Acesta este si **motivul** pentru care functia intoarce `int`, si nu `byte`, pentru a putea **semnala** sfarsitul de flux. Cu toate acestea, doar cel mai **nesemnificativ** octet al intregului va contine informatie utila.
- `int read(byte[] b)` – citeste un numar de maxim `b.length` octeti din stream si ii stocheaza in `b`. Intoarce numarul de octeti cititi sau, in caz de EOF, -1.
- `int read(byte[] b, int from, int len)` - citeste un numar de maxim `len` octeti din stream si ii stocheaza in `b`, incepand de la pozitia `from`. Intoarce numarul de octeti cititi sau, in caz de EOF, -1.
- `int available()` – intoarce numarul maxim de octeti care pot fi cititi din acest stream fara ca operatia sa se blocheze (folositoare in cazul pipe-urilor, conexiunilor internet etc).
- `int skip(long n)` - sare peste `n` octeti, intorcand numarul efectiv de octeti peste care s-a sarit, sau -1 in caz de EOF.
- `void mark(int readlimit)` - marcheaza pozitia curenta drept pozitia la care se va reveni in cazul unui apel `reset()`. Parametrul `readlimit` reprezinta dimensiunea buffer-ului in care se vor stoca octeti cititi de acum inainte. Daca se citesc mai mult de `readlimit` octeti fara a apela `reset`, atunci buffer-ul este dealocat.
- `void reset()` - se restaureaza starea stream-ului la aceea de la ultimul apel `mark`. Daca s-au citit mai multi octeti decat prevedea buffer-ul alocat, atunci se va arunca `IOException`.
- `boolean markSupported()` - returneaza adevarat daca metodele `mark` si `reset` sunt implementate, si fals in caz contrar.

Se observa ca, pentru a **defini** un *input stream* propriu, derivat direct din clasa `InputStream`, este necesar sa suprascriem **doar** metoda `read()` (singura abstracta). Toate **celelalte** metode de citire sunt deja implementate pe baza acesteia.

Ierarhia de clase derivate din `InputStream`:

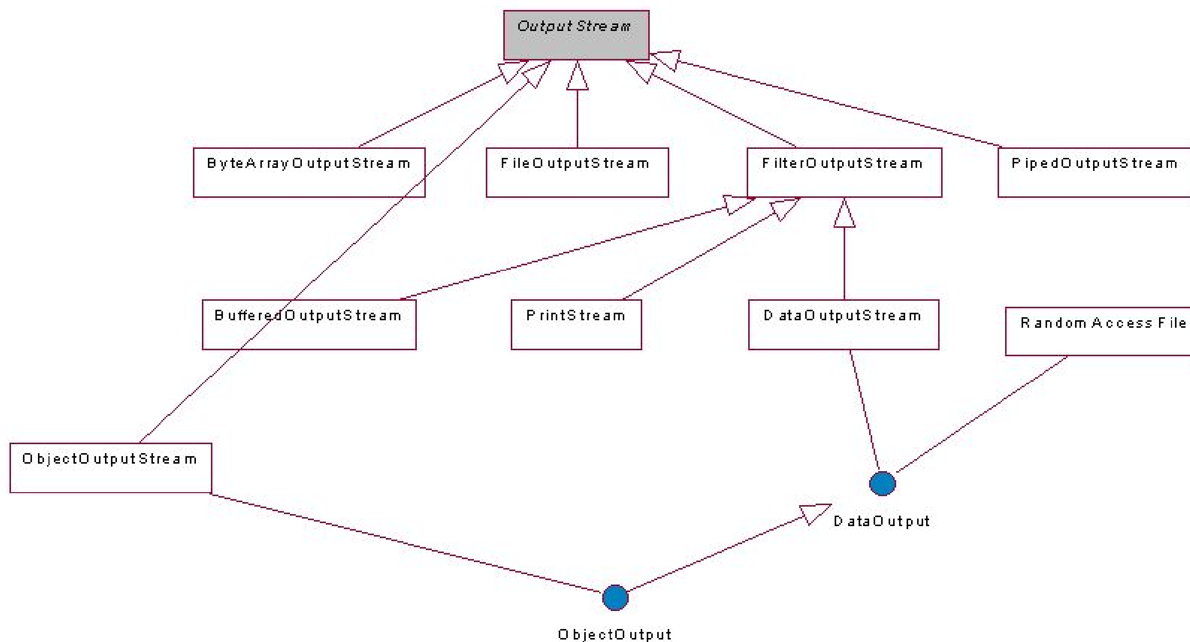


Observati clasa **`FileInputStream`**, utilizata pentru citirea dintr-un fisier sub forma unui flux de octeti.

`OutputStream`

Clasa **`OutputStream`** se situeaza in **varful** ierarhiei de clase care descriu fluxuri octet **destinatie**. Pentru **fiecare** clasa `InputStream` exista o clasa **omolog** `OutputStream`.

Operatiile din clasa `OutputStream` sunt operatiile in **oglinda** ale celor din clasa `InputStream`: `write(int b)` etc.



Mod de utilizare

Biblioteca `Java I/O` are o structura **stratificata** care permite **sporirea** "responsabilitatilor" unor obiecte individuale intr-un mod **dinamic** si **transparent**. De exemplu, am putea dori sa **imbogatim** un *input stream* de baza, care opereaza doar la nivel de octet sau sir de octeti, cu posibilitatea de a citi tipuri **primitive**: *input stream*-ul nostru ar putea citi 4 octeti deodata, pe care sa-i intoarca sub forma unui `int`.

Acest mod de a crea o structura de clase este cunoscuta sub numele de **`Decorator Pattern`**. Acest pattern impune ca obiectele care adauga functionalitate (***wrappers***) unui obiect anume sa aibe aceeasi **interfata**. Astfel, folosirea decoratorilor poate fi **transparenta**, in sensul ca putem folosi un obiect in aceeasi maniera, **indiferent** daca a fost decorat sau nu. Clasele filtru, de baza (ca `FilterInputStream`), sunt punctul de plecare pentru clasele decorator din Java I/O.

In concluzie, clasele de **baza** in ierarhia I/O **octet** devin:

- `InputStream`, `OutputStream`: pentru definirea de stream-uri de **baza**, corespunzatoare unor entitati I/O
- **`FilterInputStream`**, **`FilterOutputStream`**: pentru definirea de stream-uri **decorator**, care isi vor baza intotdeauna functionalitatea pe un **alt** stream (*underlying*), care, la randul sau, poate fi **decorat** sau nu.

Un exemplu de **decorator**, foarte des intalnit, este al claselor **`DataInputStream`**/**`DataOutputStream`**, care permit citirea de tipuri **primitive** din alte fluxuri octet, oferind metode ca `readByte`, `readInt`, `readFloat`, `readBoolean`, si omoloagele lor, `writeByte`, `writeInt` etc.

Urmatorul exemplu exemplifica citirea/scrierea unui fisier binar cu date de diferite tipuri primitive.

```
import java.io.*;

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

        DataOutputStream out = null;
        try {
            out = new DataOutputStream(new FileOutputStream("fis.meu"));
            out.writeFloat(10.0);
            out.writeInt(5);
            out.writeUTF("Hello");

        } catch(IOException e) {
            e.printStackTrace();

        } finally {
            if (out != null) {
                try {
                    out.close();

                } catch (IOException e) {
                    // error closing file: oh well...
                }
            }
        }

        DataInputStream in = null;
```

```
        try {
            in = new DataInputStream(new FileInputStream("fis.meu"));
            System.out.println("Citesc un float: "  + in.readFloat());
            System.out.println("Citesc un int: "    + in.readInt());
            System.out.println("Citesc un string: " + in.readUTF());

        } catch(IOException e) {
            e.printStackTrace();

        } finally {
            if (in != null) {
                try {
                    in.close();

                } catch (IOException e) {
                    // error closing file: oh well...
                }
            }
        }

    }
}
```

Secventa de mai sus prezinta modalitatea **preferata** de lucru cu fisiere. Observati:

- prezenta unui bloc **`finally`**, care urmareste **inchiderea** fisierului **indiferent** daca operatiile s-au incheiat cu succes sau nu
- testarea ca obiectul stream sa nu fie **`null`**, situatie in care se poate ajunge daca apelul constructorului a esuat (de exemplu, daca fisierul de citit nu a fost gasit)
- posibilitatea ca metoda **`close`** sa arunce **exceptie** (acest lucru este precizat si in documentatia metodei). In acest caz, s-a definit un bloc `try-catch` separat.

Este de notat folosirea **concomitenta** a `FileOutputStream` si `DataOutputStream`. Un `DataOutputStream` accepta date de tip **primitiv** (`writeInt`, `writeFloat`) si le trimite, mai departe, catre un `OutputStream` oarecare, pe care il primeste ca parametru in constructor. `FileOutputStream` este un `OutputStream` ce desemneaza un fisier. Pentru a scrie in **fisier** tipuri **primitive** am **decorat** un `FileOutputStream` cu un `DataOutputStream`.

In acest fel putem **imbina** diferite clase **filtru**/**decorator** pentru a obtine functionalitatea dorita. Este un fapt cunoscut ca operatiile I/O sunt mari consumatoare de **timp**. In forma lor de baza clasele I/O **nu** folosesc **buffer**-e, in sensul ca, de exemplu, fiecare operatie `write` are drept urmare o operatie **fizica** de write, pentru un stream fisier. Ar fi mult mai **convenabil** sa adunam mai **multe** date intr-un buffer si sa le scriem pe toate **deodata**. Aceasta functionalitate este oferita de clasele **`BufferedInputStream`**/**`BufferedOutputStream`**. Pentru a folosi in exemplu nostru varianta cu buffer-e adaugam inca un filtru, astfel:

```
out = new DataOutputStream(new BufferedOutputStream(new FileOutputStream("fis.meu")));
```

Stream-urile standard

Stream-urile **standard**, pe care le poseda orice program, sunt:

- `System.in`, de tip `InputStream`
- `System.out`, de tip `PrintStream`
- `System.err`, de tip `PrintStream`.

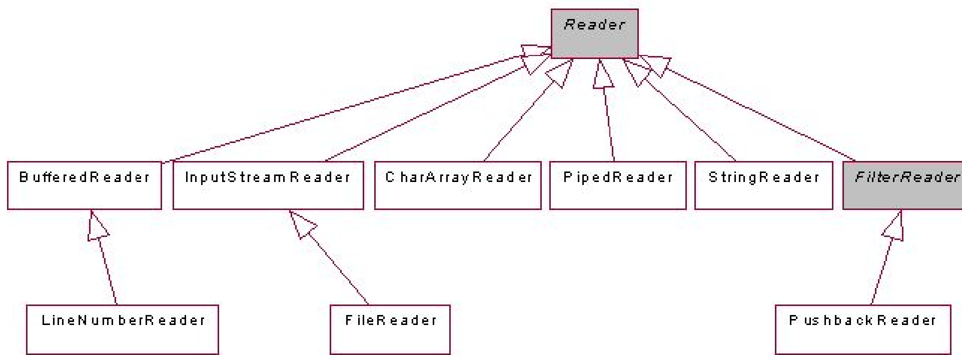
Clasa **`PrintStream`** ofera posibilitatea de formatare ca sir de caractere a valorilor avand tipuri primitive. In general, **nu** este foarte folosita. Cele 2 obiecte de mai sus si-au pastrat tipul din considerente istorice.

Stream-uri caracter

O data cu introducerea caracterelor **Unicode** a **crescut** numarul de octeti necesari pentru a reprezenta un caracter de la 1 la 2. De asemenea, a aparut notiunea de **codificare** a sirurilor (*encoding*). Astfel, a aparut necesitatea introducerii unei noi perspective asupra stream-urilor, la nivel de **caracter**.

In varful ierarhiilor claselor care lucreaza cu caractere se afla **`Reader`** si **`Writer`**. Aceste ofera primitive asemanatoare cu cele din `InputStream`/`OutputStream`, cu diferenta ca este folosit **caracterul** si nu octetul ca **unitate** de informatie. Daca dorim sa citim/scriem siruri de caractere trebuie sa folosim `Reader` si `Writer`:

Ierarhia de clase este aproape identica cu cea de la stream-uri octet:



Urmatorul exemplu arata scrierea si citirea dintr-un fisier text, linie cu linie:

```
import java.io.*;

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

        PrintWriter out = null;
        try {
            out = new PrintWriter("fis.meu");
            out.println(10.0);
            out.println(5);
            out.println("Hello");

        } catch(IOException e) {
            e.printStackTrace();

        } finally {
            if (out != null) {
                try {
                    out.close();

                } catch (IOException e) {
                    // error closing file: oh well...
                }
            }
        }

        BufferedReader in = null;
        try {
            in = new BufferedReader(new FileReader("fis.meu"));
            System.out.println("Citesc o linie: " + in.readLine());

        } catch(IOException e) {
            e.printStackTrace();

        } finally {
            if (in != null) {
                try {
                    in.close();

                } catch (IOException e) {
                    // error closing file: oh well...
                }
            }
        }

    }
}
```

Observati:

- utilizarea clasei **`PrintWriter`** pentru scrierea in fisier a valorilor avand tipuri **comune**, folosind metodele `print` si `println`
- utilizarea combinatiei **`FileReader`**/**`BufferedReader`**, pentru citirea unor **linii** din fisier, folosind metoda `readLine`. Clasa `FileReader` **nu** ofera aceasta functionalitate.

Stream-uri punte octet-caracter

Exista **conversii** de la siruri de octeti la siruri de caractere. Filtrul ce realizeaza aceasta conversie este **`InputStreamReader`**, care obtine perspectiva **caracter** asupra unui stream **octet**. `InputStreamReader` foloseste o codificare data ca parametru in constructor sau foloseste codificarea implicita.

O **aplicatie** foarte importanta a acestui mecanism o reprezinta citirea de la **consola**, tinand cont de faptul ca `System.in` are tipul

`InputStream`. Desi era, poate, mai firesc ca tipul sa fie `Reader` (orientat caracter), acesta a **ramas** orientat **octet** din motive de **compatibilitate**: in versiunea initiala nu existau stream-uri caracter.

Revenind la citirea de la consola, vom folosi, ca in exemplul din sectiunea anterioara, clasa `BufferedReader`, pentru a face uz de metoda sa, `readLine`:

```
import java.io.*;

class Test {
    public static void main(String[]arg) {
        BufferedReader in = null;
        try {
            in = new BufferedReader(new InputStreamReader(System.in));
            System.out.println("Citesc o linie: " + in.readLine());

        } catch(IOException e) {
            e.printStackTrace();

        }
    }
}
```

Acces aleator

Stream-urile au limitarea de a oferi o perspectiva exclusiv **secventiala** asupra unei surse/destinatii de date. La un moment dat, **nu** putem **sari** direct la un octet/caracter din flux, ci trebuie sa **consumam** unitatea imediat urmatoare. Este adevarat ca anumite stream-uri, cum ar fi pipe-urile sau socket-ii, nu pot fi abordati decat secvential. Pentru **fisiere** insa, situatia sta altfel: ele reprezinte obiecte care au asociat un **cursor**, ce indica pozitia urmatoarei operatii de citire/scriere, si care poate fi deplasat (***seek***).

Clasa **`RandomAccessFile`** **nu apartine** niciunei ierarhii de clase mentionate pana acum. Ea ofera posibilitatea **deplasarii** prin fisierul deschis, folosind `seek` si implementeaza functionalitatea `DataInput` si `DataOutput` (cea implementata si in `DataInputStream`, `DataOutputStream`), pentru a putea lucra cu tipuri **primitive**. Este asemanatoare ca functionalitate cu folosirea fisierelor in ANSI C (`fseek`, `fread`, `fwrite` etc).

Iata un exemplu simplu, asemanator celor de la stream-uri:

```
import java.io.*;

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

        RandomAccessFile out = null;
        try {
            out = new RandomAccessFile("fis.meu", "rw"); // read and write
            out.writeFloat(10.0);
            out.writeInt(5);
            out.writeUTF("Hello"); // these methods are also in DataOutputStream, since both classes implement 
DataOutput interface

        } catch(IOException e) {
            e.printStackTrace();

        } finally {
            if (out != null) {
                try {
                    out.close();

                } catch (IOException e) {
                    // error closing file: oh well...
                }
            }
        }
    }
}
```

Manipularea fisierelor

Spre deosebire de clasele mentionate pana acum, care gestioneaza **continutul** fisierelor, clasa **`File`** permite manipularea fisierelor ca **entitati**, oferind o reprezentare abstracta a cailor de **fisiere** si **directoare**. Ea ofera operatii ca:

- `boolean createNewFile()` – creaza un nou fisier, la calea data in constructor
- `boolean delete()` - sterge fisierul
- `boolean exists()` – verifica daca fisierul/directorul dat de cale exista
- `boolean isDirectory()` – intoarce `true` atunci cand calea denota un director

- boolean isFile() - intoarce true atunci cand calea denota un fisier
- long length() – intoarce dimensiunea fisierului
- String[] list() – intoarce intrarile din director

Spre exemplu, pentru a afisa continutul directorului curent vom proceda astfel:

```
import java.io.*;

public class DirList {
    public static void main(String[] args) {
        File path = new File("."); // current directory
        String[] list = path.list();

        for(int i = 0; i < list.length; i++)
            System.out.println(list[i]);
    }
}
```

Exercitii

1. (**2p**) Implementati comanda UNIX cp, care realizeaza **copierea** unui fisier.
   - Sintaxa de **apelare**: java Cp <sursa> <destinatie>.
   - Programul va putea copia **orice** tip de fisier (binar sau text). Care este solutia **universala** pentru a implementa acest comportament?

     Hint: pentru stabilirea, din Eclipse, a parametrilor in linia de comanda, executati: click dreapta pe proiect -> Run As -> Run Configurations... -> tab-ul Arguments.
2. (**2p**) Implementati comanda UNIX wc (word count), care determina numarul de **cuvinte** dintr-un fisier text.
   - Sintaxa de **apelare**: java Wc <sursa>.
   - Se va afisa numarul de cuvinte din fisier.
   - Daca primeste **switch**-ul -l (parametru in linia de comanda), trebuie sa numere **liniile**.
3. (**2p**) Implementati comanda UNIX grep, care **cauta** un cuvant intr-un fisier text.
   - Sintaxa de **apelare**: java Grep <sursa> <cuvant>.
   - Se va afisa **numarul** liniei si **continutul** acesteia.
4. (**3p**) Implementati clase pentru input si ouput de date **criptate**.
   - Prin **criptarea** unui octet intelegem **incrementarea** acestuia cu o valoare.
   - Cele doua clase vor fi **decoratori** de nivel **octet**, construiti pe baza **altor** stream-uri
     - Stream-ul de **iesire** cripteaza si deleaga fiecare octet primit stream-ului de baza, dupa ce, in prealabil, il transforma.
     - Asemanator in cazul stream-ului de **intrare** (decriptare).
   - Modificati programul Cp, astfel incat **copierea** sa se faca direct in forma **criptata**.
   - Modificati programul Wc, astfel incat **numararea** sa se faca pentru un fisier **criptat**.
5. (**2p**) Implementati un utilitar, similar comenzii UNIX ls:
   - daca parametrul este un **fisier**, se vor afisa numele si dimensiunea acestuia
   - daca parametrul este un **director**, se vor afisa numele si dimensiunea tuturor fisierelor si directoarelor continute. Este necesara scrierea unei functii care calculeaza, **recursiv**, dimensiunea unui director deoarece, implicit, dimensiunea acestuia este 0.
   - Testati pentru directorul parinte (..).
6. (**1p**) Fisierul secret.txt este **binar** si contine la inceput un long. Acesta reprezinta offset-ul unui intreg din fisier. Aflati valoarea intregului!
   - Care este cea mai potrivita clasa pentru aceasta problema?

- **Solutii**

Referinte

- I/O pe Java Tutorials

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

Vizualizări

- Pagină
- Discuție
- Vezi sursa
- Istoric

Unelte personale

- Autentificare

Navigare

- Pagina principală
- Portalul comunităţii
- Discută la cafenea
- Schimbări recente
- Pagină aleatorie
- Ajutor

Caută

Du-te Caută

Trusa de unelte

- Ce se leagă aici
- Modificări corelate
- Trimite fişier
- Pagini speciale
- Versiune de tipărit
- Legătură permanentă
- Print as PDF

- Ultima modificare 17:07, 30 decembrie 2010.
- Această pagină a fost vizitată de 10.033 ori.
- Conţinutul este disponibil sub GNU Free Documentation License 1.2.
- Politica de confidenţialitate
- Despre POO
- Termeni