Intrări / Ieșiri în Java. Fișiere

Java Input / Output. Files.

Obiective:
- înțelegerea mecanismelor de I/O;
- lucrul cu fișiere text și binare

Objectives:
- understanding the I/O mechanisms;
- working with text and binary files;

Prin flux (stream) înţelegem un canal de comunicaţie unidirecţional între două procese. Un flux care citeşte date se numeşte flux de intrare iar altul care scrie date se numeşte flux de ieşire.

La nivel inferior un stream asigură transmiterea/recepţia datelor pe 8 biţi (byte-octet). Un fişier este o colecţie de articole memorate pe un dispozitiv extern cum ar fi floppy disk, CD-ROM, etc. Accesul fişierelor în Java se face prin clase de tip *FileStream*, ce furnizează mecanismul de acces, dar nu păstrează conţinutul lor. Acest nivel inferior bazat pe stream-uri permite transmiterea pe 8 biţi, dar oferă funcţionalităţi referitoare la procesarea la nivel superior. De exemplu, metoda *readInt()* va procesa 4 grupuri de 8 biţi pentru că o variabilă întreagă e stocată pe 32 de biţi în Java.

La Intrări/Ieşiri, I/I, pe octet există 2 sisteme paralele având ca şi rădăcină clasele *InputStream* şi *OutputStream*.

A doua variantă oferă clasele *Reader* şi *Writer* pentru manipulări de caractere text de tip Unicode pe 16-32 biţi şi UTF (Universal Character Set Transformation Format).

Toate clasele Java referitoare la intrări şi ieşiri sunt situate în pachetul *java.io*.

https://docs.oracle.com/javase/8/docs/api/java/io/***.html

unde *** se înlocuiește cu numele clasei căutate.

Principalele clase derivate din *InputStream* sunt ierarhic prezentate astfel:

- *ByteArrayInputStream*
- *FileInputStream*
- *PipedInputStream*
- *ObjecttInputStream*
- *SequenceInputStream*
- *FilterInputStream* care la rândul ei are următoarele subclase:
    - *DataInputStream*
    - *BufferedInputStream*
    - *LineNumberInputStream*
    - *PushBackInputStream*

Clasa *InputStream* este o clasă abstractă, fiind aşa cum se vede mai sus părinte pentru alte clase, care deci vor implementa în mare acelaşi comportament. Metodele membre principale sunt prezentate mai jos.

- *int read () throws IOException*

=> citeşte un byte

- *int read (byte [] buffer) throws IOException*

=> citeşte un tablou de valori

- *long skip (long n)*

=> sare peste numărul de valori indicat de la intrare

- *int available () throws IOException*

=> determină nunarul de octeţi care pot fi citiţi fără a fi grupaţi în blocuri

- *void close () throws IOException*

=> închide stream-ul de intrare

Clasele ce se referă la citirea datelor dintr-un tablou de bytes, fişier sau pipe sunt legate de *surse de intrare fizice*.

Intrările virtuale, sunt realizate cu celelalte clase şi depind de un alt stream de intrare după care acţionează extinzând funcţionalitatea într-un anumit mod. De exemplu, clasa *PushBackInputStream* citeşte valori de la un alt stream de intrare, dar de asemenea permite programului să "nu citească" caractere pe care le pune înapoi în stream-ul de intrare. În mod real aceste caractere ce trebuie puse înapoi vor fi puse într-un buffer local. Când va mai apărea o operaţie de citire caracterele vor fi întâi citite din acest buffer.

Stream-uri de intrare fizice şi virtuale

Clasele principale de tipul stream-urilor de intrare fizice se disting prin numele lor şi argumentele folosite în cadrul constructorilor. Prototipurile principalilor constructori sunt:

- *ByteArrayInputStream (byte [] buffer)*
- *ByteArrayInputStream (byte [] buffer, int offset, int count)*
- *FileInputStream (File f)*
- *FileInputStream (String fileName)*
- *PipedInputStream (PipedOutputStream p)*

E de remarcat că doar pentru simple citiri sau scrieri un *FileInputStream* sau *FileOutputStream* poate fi manipulat fără a crea un obiect de tip *File*. În general un obiect de tip *File* este necesar doar când obiectul efectuează operaţii asupra fişierului însuşi cum ar fi redenumirea lui sau ştergerea lui. Un alt motiv pentru crearea unui obiect de tip *File* este pentru testarea faptului dacă un fişier este citibil sau apt de a se scrie în el înainte că secvenţa de date să sosească de /la intrare/ieşire.

Clasele *SequenceInputStream, ObjectInputStream* şi subclasele lui *FilterInputStream* pot fi considerate ca şi clase de intrare virtuale. Nici una din ele nu citeşte caractere dintr-o zonă de intrare, de aceea ele sunt legate de unul sau mai multe stream-uri de intrare privind sursele lor de date. Fiecare adaugă funcţionalităţi specifice clasei. De exemplu, *SequenceInputStream* preia o secvenţă de două sau mai multe stream-uri de intrare şi în mod logic le plasează unul după altul, cap la coadă.
Când un stream de intrare este epuizat următorul stream din secvenţa e pornit fără întreruperi şi fără a fi nevoie de vreo acţiune din partea utilizatorului. Stream-urile fundamentale pot fi specificate ca două argumente în cadrul constructorului sau ca o enumerare la o colecţie de stream-uri de intrare (dacă sunt mai mult de 2 stream-uri).

Clasa *ObjectInputStream* este folosită în procesul de serializare, adică are abilitatea de a converti valorile unui obiect într-o reprezentare care poate fi transmisă ca şi o secvenţă de valori pe 8 biţi. Ea permite reconstituirea unui obiect care a fost serializat şi transmis într-un flux. Citirea se face cu metoda *readObject()*. Obiectele ce vor fi serializate sunt instanțe ale unor clase care implementează interfaţa *Serializable* sau *Externalizable*.

Stream-uri de ieşire

Subclasele clasei *OutputStream* oferă aceleaşi utilităţi datorită polimorfismului, compoziţiei şi moştenirii, ca şi *InputStream*. Principalele metode declarate de această clasă sunt:

- *public abstract void write (int b) throws IOException*

=> scrie un octet

- *public void write (byte [] buffer, int offset, int length) throws IOException*

=> scrie *length* octeţi din şirul *buffer*, începând cu poziţia *offset*

Principalele subclase derivate din *OutputStream* sunt:

- *ByteArrayOutputStream*
- *FileOutputStream*
- *PipedOutputStream*
- *ObjectOutputStream*
- *FilterOutputStream care la rândul ei conţine:*
  - *DataOutputStream*
  - *BufferedOutputStream*
  - *PrintStream*

În principiu *OutputStream* permite scrierea byte cu byte la ieşire. Pentru a mări eficiența scrierii majoritatea claselor lucrează cu buffere pentru a realiza această operaţie.

Stream-urile de ieşire pot fi şi ele împărţite în două categorii:

- cele care caracterizează locaţia fizică a ieşirii
- cele care adăugă un comportament mai mare la stream-ul de ieşire.

În primul grup avem clasele:

- *ByteArrayOutputStream*, scrie într-un tablou
- *FileOutputStream*, scrie într-un fişier
- *PipedOutputStrem*, scrie într-un pipe

În al 2-lea grup avem:

- *ObjectOutputStream* care poate scrie la ieşire obiecte serializate
- *FilterOutputStream* care adăugă noi funcţionalităţi la o operaţie de ieşire. Un filtru de ieşire execută anumite sarcini înainte de a trimite valorile către stream-ul de ieşire. Avem următoarele subclase principale pornind de la *FilterOutputStream* şi anume:
- *BufferedOutputStream*, ce menţine un buffer intern de valori care va fi scris doar când bufferul este plin sau ieşirea este golită (*flushed*)
- *DataOutputStream*, adaugă metode pentru a scrie valorile binare pentru fiecare tip de data primitivă
- *PrintStream* e similar dar generează o reprezentare textuală pe ecran faţă de cea binară. Ea este utilizată pentru a genera ieşiri care vor fi citite de utilizatorii umani şi nu ca o intrare pentru un alt program.

System.out şi *System.err* sunt instanțe ale lui *PrintStream*. Principalele metode ale clasei *PrintStream* sunt prezentate mai jos:

- *public void print (boolean bool)*

=> scrie o valoare de tip *boolean*

- *public void print (int inum)*

=> scrie o valoare de tip *int*

- *public void print (float fnum)*

=> scrie o valoare de tip *float*

- *public void print (double dnum)*

=> scrie o valoare de tip *double*

- *public void print (String str)*

=> scrie o valoare de tip *String*

- *public void print (Object obj){*
  printț (obj.toString());
  }

=> scrie reprezentarea textuala a unui obiect

Accesul prin stream folosind *Stream.out* va fi în continuare o instanţă a lui *PrintStream* pentru că prea mult cod în Java depinde deja de aceasta. Crearea de noi date de tip stream folosind *PrintStream* este descurajată datorită apariţiei noii clase *PrintWriter* cu facilităţi similare şi care mai suporta şi caractere pe 16 biţi de tip Unicode.

Clasa *ObjectOutputStream* permite serializarea obiectelor, adică convertirea unui obiect într-o reprezentare care se poate adapta la gruparea în bytes a informaţiei. Ea a fost proiectată pentru scrierea valorilor unui obiect într-un stream într-o formă ce permite ca să fie uşor citite înapoi cu *ObjectInputStream*.

*Reader*s şi *Writer*s

Ierarhiile claselor pornind de la clasele *Reader* şi *Writer* în mare oglindesc funcţionalităţile oferite de clasele *InputStream* şi *OutputStream* şi dependinţele lor.

Reader-ele şi *Writer*-ele manipulează caractere Unicode pe 16 biţi. Ierarhiile acestor clase sunt următoarele:

- *Reader* din care avem:
  - *BufferedReader* din care avem:
    - *LineNumberReader*
  - *CharArrayReader*
  - *InputStreamReader* din care avem:
    - *FileReader*
  - *FilterReader* din care avem:
    - *PushBackReader*
  - *PipedReader*
  - *StringReader*

respectiv

- *Writer* din care avem:
  - *BufferedWriter*
  - *CharArrayWriter*
  - *OutputStreamWriter* din care avem:
    - *FileWriter*
  - *FilterWriter*
  - *PipedWriter*
  - *PrintWriter*
  - *StringWriter*

Ca şi la stream-urile de intrare putem avea următoarea clasificare:

- cele care manipulează direct datele: *CharArrayReader*, *StringReader, FileReader*
- cele care adăugă noi funcţionalităţi datelor generate de alţi Reader: *BufferedReader, LineNumberReader, FilterReader.*

Reader-ele şi *Writer*-ele sunt utile atunci când lucram cu date pur textuale şi nu cu date binare cum este cazul imaginilor sau a culorilor.

Lucrul cu fişiere în Java

Un fişier este o colecţie de înregistrări păstrate de regulă pe un suport extern şi identificată printr-un nume. Fiecare înregistrare este o grupare de informaţii sau de date care poate fi tratată în mod unitar.

În limbajul Java, fişierul este privit ca sursa sau destinaţia unui flux de date. În cazul citirii din fişier, informaţiile se transmit de la acesta către memoria internă sub forma unui flux de intrare. În cazul operaţiei de scriere, datele se transmit de la memoria internă la fişier sub forma unui flux de ieşire. Comunicarea între memoria internă şi un fişier de text se face sub forma unui flux de caractere. Pe platformele pe care reprezentarea caracterelor se face pe un octet (de exemplu în cod ASCII), acesta este tratat ca şi un flux de octeţi.

În cazul fişierelor de date, comunicarea dintre memoria internă şi fişier se poate face prin fluxuri de caractere numai dacă datele din fişier sunt reprezentate exclusiv în format extern (deci sub formă de şiruri de caractere). Dacă însă există şi câmpuri de date în format binar, legătura dintre memoria internă şi fişierul de date se face prin fluxuri de octeţi.

Pentru a lucra cu un fişier, se efectuează următoarele operaţii:

1) se deschide fişierul, scop în care trebuie să se comunice programului numele fişierului, locația în care se găseşte (de exemplu unitatea de disc şi directorul) şi modul în care va fi utilizat (pentru citire, pentru scriere sau în ambele moduri);
2) se procesează fişierul, efectuând o succesiune de operaţii de citire/scriere;
3) se închide fişierul.

Clasa *File*

Instanţele clasei *File* conţin informaţii privind numele fişierului şi calea pe care se găseşte acesta. Clasa *File* oferă, de asemenea, metode prin care se pot face unele operaţii legate de prezenţa fişierului respectiv: se poate afla dacă fişierul există, dacă el poate fi citit sau scris, se poate crea un fişier nou, se poate şterge un fişier existent etc. *Calea* indica modul în care poate fi localizat fişierul de pe disc. Calea poate fi absolută sau relativă. Instanţele clasei *File* sunt căile fişierelor.

Clasa *FileInputStream*

Clasa *FileInputStream* permite citirea datelor din fişiere sub formă de fluxuri de octeţi. Orice instanţă a acestei clase este un flux de intrare, care are ca sursa un fişier. La crearea acestei instanțe se cauta şi se deschide fişierul indicat că argument al constructorului. Dacă fişierul nu există, sau nu poate fi deschis pentru citire, se generează o excepţie.
Metodele acestei clase permit să se citească din fişierul de intrare octeţi sau secvenţe de octeţi, fără a le da nici o interpretare.

Clasa *FileReader*

Citirea unui fişier de text se poate şi cu o instanţă a clasei *FileReader*. Deosebirea este că această ultimă clasă crează un flux de caractere, în loc de un flux de octeţi. Chiar dacă fişierul nu este codificat în Unicode, ci în alt cod de caractere (de cele mai multe ori ASCII), se face automat conversia în Unicode. Clasa *FileReader* este derivată din clasa *InputStreamReader* şi foloseşte metodele acesteia.

Clasa *FileOutputStream*

Fiecare instanţă a clasei *FileOutputStream* este un flux de octeţi de ieşire conectat la un fişier, în care se scriu octeţii primiţi din flux. Fluxul se poate conecta însă şi la un alt flux de octeţi de ieşire deja existent.

Clasa *FileWriter*

Scrierea într-un fişier de text se poate face, de asemenea, folosind clasa *FileWriter*. Instanţele acestei clase sunt fluxuri de caractere de ieşire, prin care se face scrierea într-un fişier. Clasa *FileWriter* este derivată din *OutputStreamWriter* şi foloseşte metodele acesteia.

Clasa *RandomAccessFile*

Clasa *RandomAccessFile* este derivată direct din clasa *Object*, deci nu face parte din niciuna din ierarhiile de fluxuri de intrare/ieşire, dar aparţine pachetului *java.io*.

Fişierul cu acces direct este privit aici ca un tablou de octeţi memorat într-un sistem de fişiere. Există un cursor al fişierului, care se comportă ca un *indice* al acestui tablou. Valoarea acestui indice poate fi citită cu metoda *getFilePointer()* şi poate fi modificată cu metoda *seek().* Orice citire sau scriere se face începând de la acest cursor. La sfârşitul operaţiei, cursorul se deplasează pe o distanţă corespunzătoare cu numărul de octeţi care au fost citiţi sau scrişi efectiv.

Fişierul cu acces direct poate fi deschis în modul *read* (numai pentru citire), *write* (numai pentru scriere) sau *read/write*. În ultimul caz, este posibil să se efectueze atât citiri cât şi scrieri.

Scrierea în fişierele cu acces direct se poate face atât în mod text cât şi în mod binar.

Lucru individual

1. Se citesc de la tastatură date formatate sub forma DD/MM/YYYY. Să se afișeze sub forma DD luna YYYY, unde luna este forma expandată a MM și de asemenea să se afișeze și dacă anul este bisect. Programul se oprește prin apăsarea tastei X

==========

2. Scrieți o aplicație Java ce citește un set de fișiere text ce conțin informațiile studenților din anul curent. Fișierele sunt stocate pe sistemul local cu denumirea An_y_Grupa_xxxx.txt . Să se agregheze informația din aceste fișiere folosind *SequenceInputStream* și să se genereze un nou fișier ce conține toți studenții din anul curent ordonați alfabetic.

==========

3. Se dă un fișier *.csv ce conține următoarele câmpuri separate prin simbolul /: nume, prenume, număr de telefon, data nașterii, link la profilul de Facebook. Să se citească informația din fișier și să se genereze noi fișiere (individuale) ce conțin doar persoanele cu următoarele caracteristici: persoanele născute în luna decembrie, persoane ale căror numere de telefon sunt externe României sau au număr de telefon fix, persoane cu numele Andrei sau Nicolae și persoane ale căror link-uri de la profilul de Facebook nu au fost customizate (conțin un șir aleator de numere la finalul acestuia).

==========

4. Se dă un fișier binar ce conține o secvență de caractere ce reprezintă o cheie privată pentru Bitcoin (256 caractere). De la tastatură se citește o secvență de caractere ce reprezintă cheia publică a unui Bitcoin. Să se genereze identificatorul de tranzacție corespunzător acestor informații folosind operația de *și exclusiv (XOR)* pe biți. Scrieți informația rezultată într-un fișier binar.

==========

5. Scrieți o aplicație Java ce citește un fișier cu următorul format:

*/rnd1_001.lab

A 0001

C 0003

D 0004

F 0003

A 0006

.

*/rnd2_002.lab

C 0003

F 0001

Z 0010

M 0011

.

…..

Să se separe informația din fișier în fișiere distincte denumite conform liniei ce începe prin caracterele */.

==========

6. Citiți dintr-un fișier text o imagine grayscale reprezentată ca o matrice de întregi. Imaginea

este urmată de mai multe filtre de convoluție. Aplicați aceste filtre pe imaginea originală și afișați

atât imaginea originală cât și imaginile rezultate.

Exemple de filtre de convoluție:

Blur:

1 2 1

2 4 2

1 2 1

Sharpen:

-1 -1 -1

-1 9 -1

-1 -1 -1

Edge detection:

-1 -1 -1

-1 8 -1

-1 -1 -1

==========

7. Scrieți o aplicație Java ce permite serializarea unui obiect ce reprezintă un algoritm de sortare (la alegere). Citiți obiectul serializat și aplicați-l asupra unui șir de numere citit de la tastatură. Numerele sunt citite pe rând, iar sfârșitul șirului este marcat prin apăsarea oricărei taste ce nu reprezintă un număr.

Individual work

1. Read from the keyboard strings representing dates formated as DD/MM/YYYY. Print the dates as DD month YYYY, where month is the expanded version of the MM, and also if the year is a leap year. The program exits when the user types in "X" .

==========

2. Write a Java application which reads a set of text files which contain student information from the current study year. The files are stored locally under the names Year_Y_Group_XXXX.txt . Agregate the information in this file using a *SequenceInputStream* and generate a new file which contains all the students ordered alphabetically.

==========

3. You are given a *.csv file which contains the following fields separated by the "/" symbol: name, surname, phone number, date of birth, link to Facebook profile. Read the information in the file and generate individual files containing the following information: people born in December, people whose phone numbers are international (not Romanian) or are landline numbers, people named Andrei and Nicolae and people whose Facebook profile link is not customised (have a random sequence of digits at the end of the link).

==========

4. You are given a binary file which contains a sequence of characters representing a private Butcoin key (256 characters). From the keyboard read a sequence of characters which represents the public key for a Bitcoin. Generate the transaction id for this information by using the XOR bitwise operator. Write the resulting number in binary file.

==========

5. Write a Java application which reads a file with the following format:

*/rnd1_001.lab

A 0001

C 0003

D 0004

F 0003

A 0006

.

*/rnd2_002.lab

C 0003

F 0001

Z 0010

M 0011

.

…..

Separate the information from the file into distinct files which are named according to the line which starts with */ .

==========

6. Read from a text file a grayscale image represented as an array of integer values. The image is followed by multiple convolution filters. Apply these filters to the original image and display both the original image, and the filtered results. Examples of convolution filters:

Blur:

1 2 1

2 4 2

1 2 1

Sharpen:

-1 -1 -1

-1 9 -1

-1 -1 -1

Edge detection:

-1 -1 -1

-1 8 -1

-1 -1 -1

==========

7. Write a Java application which enables the serialization of an object representing a sorting algorithm (your choice). Read the serialized object and apply the algorithm to a sequence of numbers read from the keyboard. Each number is read separately and the end of the sequence is marked by typing any key which does not represent a number.