Java Input / Output. Files.

Objectives:

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

A data stream means a one-way communication channel between two processes/devices. A flow that reads data is called an input stream, and a flow that writes data is called an output stream.

All Java classes related to input and output operations are in the packages *java.io* și *java.nio.*
https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/io/package-summary.html
https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/nio/package-summary.html

Low level streams handle the transmission/reception of data on 8 bits (byte).
High level streams handle the transmission/reception of formatted data (primitives or class instances).
Fluxurile de date de tip `Reader` şi `Writer` tratează transmiterea/recepția datelor la nivel text (Unicode pe 16-32 biţi şi UTF).
The files are accessed using `FileStream` classes.

Physical streams process data related to files, pipes, or physical devices.
Virtual streams process data related to physical streams.

The hierarchy of Input/Output classes is based on `InputStream` şi `OutputStream` classes.

Input streams

The main classes derived from `InputStream` are in the following hierarchy:

- `ByteArrayInputStream`
- `FileInputStream`
- `PipedInputStream`
- `ObjectInputStream`
- `SequenceInputStream`
- `FilterInputStream`
    - `DataInputStream`
    - `BufferedInputStream`
    - `LineNumberInputStream`
    - `PushBackInputStream`

The main methods of the `InputStream` class (inherited in its subclasses):

- `int read () throws IOException`

=> reads a byte

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

=> reads an array of bytes

- `long skip (long n)`

=> skips the number of values received as parameter

- `int available () throws IOException`

=> determines the number of bytes that can be read without being grouped into blocks

- `void close () throws IOException`

=> closes the input stream

Output streams

The main methods of the `OutputStream` class:

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

  => writes a byte
- `public  void write (byte [] buffer, int offset, int length) throws IOException`

  => writes *length* bytes from the *buffer* array starting at the *offset* position

The main subclasses derived from `OutputStream` are:

- `ByteArrayOutputStream`
- `FileOutputStream`
- `PipedOutputStream`
- `ObjectOutputStream`
- `FilterOutputStream`
  - `DataOutputStream`
  - `BufferedOutputStream`
  - `PrintStream`

and they have specific methods for maneuvering the data.

***Reader*s and *Writer*s**

The hierarchies of the classes inherited from `Reader` and `Writer` largely mirror the functionalities offered by the `InputStream` and `OutputStream` classes and their dependencies.

- `Reader:`
  - `BufferedReader:`
    - `LineNumberReader`
  - `CharArrayReader`
  - `InputStreamReader:`
    - `FileReader`
  - `FilterReader:`
    - `PushBackReader`
  - `PipedReader`
  - `StringReader`

- `Writer:`
  - `BufferedWriter`
  - `CharArrayWriter`
  - `OutputStreamWriter:`
    - `FileWriter`
  - `FilterWriter`
  - `PipedWriter`
  - `PrintWriter`
  - `StringWriter`

As with the input streams we can have the following classification:

- streams that directly manipulate the data : `CharArrayReader, StringReader, FileReader`
- streams that add new functionalities to the basic readers: `BufferedReader,  LineNumberReader, FilterReader`, etc.

Working with files in Java

A file is a collection of records typically kept on an external media and identified by a name. Each record is a grouping of information or data that can be treated in a consistent manner.

In Java, a file is viewed as the source or destination of a data stream.
In case of reading from the file, the information is transmitted from the file to the internal memory in the form of an input stream.
In the case of the write operation, the data is transmitted from the internal memory to the file in the form of an output stream.

The data to be read/written from/to a file can be transferred as text or in binary format.

To work with a file, the following operations are performed:
1) the file is opened (path and opening mode are specified)
2) the file is processed by performing a series of read/write operations
3) the file is closed

Class `File`

`File` class instances contain information about the file name and its path (absolute or relative). The `File` class also provides methods that do some operations related to the presence of that file: the file existance can be checked, whether it can be read or written, a new file can be created, an existent file can be deleted, etc.

Class `FileInputStream`

The `FileInputStream` class allows reading files in byte array format. Any instance of this class is an input stream, which has a file as data source. If the file does not exist, or cannot be opened for reading, an exception is generated.

Clasa `FileReader`

The `FileReader` class serves at reading text input files. It creates a character stream instead of a simple byte array stream. Even if the file is not encoded in Unicode but in another character representation (most often ASCII), it is automatically converted to Unicode. The `FileReader` class is derived from the `InputStreamReader` class and uses its methods.

Clasa `FileOutputStream`

Each instance of the `FileOutputStream` class represents a stream of output bytes connected to a file. However, the stream can also be connected to another existing byte output stream.

Clasa `FileWriter`

The instances of this class are output character streams, serving at writing text data to a file. The `FileWriter` class is derived from `OutputStreamWriter` and uses its methods.

Clasa `RandomAccessFile`

The `RandomAccessFile` class is derived directly from the `Object` class, so it is not part of any of the input/output interitance hierarchies, but belongs to the `java.io` package.

This class allows maneuvering the position indicator attached to a file. The value of this index can be read with the `getFilePointer()` method and can be changed with the `seek()` method. Any reading or writing is controlled by this file indicator. At the end of each I/O operation, the pointer moves over a distance corresponding to the number of bytes that were actually read or written.

The direct access file can be opened in *read-only*, *write-only* or *read/write* mode.

Reading and scribbling in direct access files can be done in both text and binary mode.

Examples

Ex. 1 – text I/O streams; text files

```
import java .io.* ;

public class FileReaderWriter_01{

      public static void main (String[] args){
           String s1,s2=new String();
           String str=new String();
           String str2=new String();

           try{
      BufferedReader bufRead=new BufferedReader (new InputStreamReader
(System.in));
                System.out .println ("Prima linie?");
                str=bufRead.readLine ();
                System.out .println("Linia scrisa este: "+str);
                System.out .println("A doua linie?");
                str2=bufRead.readLine ();
                System.out .println("Linia scrisa este: "+str2);
                bufRead.close ();
           }
           catch(IOException e){System.out .println("Eroare citire" +e);}

           try{
      PrintWriter  outF1=new  PrintWriter  (new  BufferedWriter  (new  FileWriter
("test.dat")));
                outF1 .println(str);
                outF1.close ();

           BufferedReader inF1=new BufferedReader (new FileReader ("test.dat"));
                while((s1=inF1.readLine ())!=null)
                     s2 +=s1 + "\n";
                System.out .println("Citire din fisier: "+s2);
                inF1.close();

                PrintWriter outF2=new PrintWriter (new FileWriter ("test.dat"));
                outF2.println("Asta este altceva");
                outF2.close();

           BufferedReader inF2=new BufferedReader (new FileReader ("test.dat"));
                while((s1=inF2.readLine ())!=null)
                     s2 +=s1 + "\n";
                System.out .println("Citire din fisier: "+s2);
                inF2.close();
                System.out .println("Citire BufferedReader-StringReader");
                BufferedReader inBS=new BufferedReader (new StringReader (str2));
                System.out .println(inBS.readLine());
           }
           catch(IOException e){System.out .println("Exceptie: "+e);}

           try{
                int c;
                StringReader inSR1=new StringReader (s2);
                System.out .println("Citire cu StringReader cu conversie in char");
                while((c=inSR1.read()) != -1)
                     System.out .print((char)c);
                inSR1.close ();
           System.out .println("Citire cu StringReader fara conversie in char");
                StringReader inSR2 =new StringReader (s2);
                while((c = inSR2.read()) != -1)
                     System.out .print(c);
                inSR2.close();
           }
           catch (IOException e){System.out .println("Eroare citire din memorie");}

      }

}
```

Ex. 2 – binary files

```
import java.io .* ;
public class FileInputOutputStream_02{

           public static void main (String[] args){
           File inputFile = new File("FileInputOutputStream_02.jpg");
           File outputFile = new File("FileInputOutputStream_02_copy.jpg");

           byte buf[]=new byte[1024];
           try{
                FileInputStream fis = new FileInputStream(inputFile);
                FileOutputStream fos = new FileOutputStream(outputFile);

                while(fis.read(buf) != -1)
                     fos.write(buf);

                fis.close();
                fos.close();
                }
           catch(IOException e){
                System.out .println("Eroare: "+e.toString    ());
           }

           System.out .println("Copiere terminata...");
           }
}
```

Ex. 3 – serialization, deserialization

```
import java .io.* ;
```

```
class MyVector implements Serializable {
      private static final long serialVersionUID = 1L;
      int dim=3;
      private int[] tab = new int[dim];
      public MyVector(){
           tab[0]=2;
           tab[1]=1;
      }

      public void scrieVector(int i,int j){
           tab[i]=j;
      }
      public int citesteVector(int i){
           return tab[i];
      }
}

public class ObjectOutputStream_01{
      public static void main (String[] args){
           String obj1 = "Sir de caractere";
           MyVector obj2 = new MyVector();
           try{
                FileOutputStream fisIesire=new FileOutputStream("temp.tmp");
      ObjectOutputStream obIesireSerie=new ObjectOutputStream (fisIesire);
                obIesireSerie.writeObject(obj1);
                System.out .println("Vector initial: "+
                " tab[0]= "+ obj2.citesteVector(0)+ " tab[1]= " +
                obj2.citesteVector(1)+" tab[2]= "+
                obj2.citesteVector(2));
                obIesireSerie.writeObject(obj2);

                obIesireSerie.close ();
           }
           catch(IOException e) {
                System.out .println ("Eroare la serializare!");
           }

           System.out.println("S-a terminat serializarea");
           System.out.println("Apasa ENTER pt. deserializare");

           try{System.in.read ();}catch(IOException e){}

           try{
                FileInputStream fisIntrare=new FileInputStream ("temp.tmp");

                ObjectInputStream obIntrareSerie=new ObjectInputStream
                (fisIntrare);

                String obInS1=(String)obIntrareSerie.readObject();
                System.out .println(obInS1);

                MyVector vctRezultat=(MyVector)obIntrareSerie.readObject ();
                System.out .println("Vector citit serial:"+
                " tab[0]= "+vctRezultat.citesteVector(0)+
                " tab[1]= "+vctRezultat.citesteVector(1)+
                " tab[2]= "+vctRezultat.citesteVector(2));
                vctRezultat.scrieVector(1,5);

                System.out .println("Vector preluat modificat:"+
                " tab[0]= "+vctRezultat.citesteVector(0)+
                " tab[1]= "+vctRezultat.citesteVector(1)+
                " tab[2]= "+vctRezultat.citesteVector(2));

                fisIntrare.close();
            }
            catch(IOException e){
                System.out .println("Eroare la deserializare");
            }
            catch(ClassNotFoundException e){
                System.out .println("ClassNotFoundException!!!");
            }
        }
}
```

Ex. 4 – random access files

```
import java .io .* ;
public class RandomAccessFile_01{
            static int dim1=10;
            static int dim2=64;
            public static void main (String[] args){
            byte[] bufRead=new byte[dim1];
            byte[] bufWrite;
            String temp;
            int conv=0;
            byte buf[]=new byte[dim2];
            try{
                System.out.println("Scrie un text: ");
                System.in.read (buf,0,dim2);
            }
            catch (IOException e) {
                System.out .println("Eroare: "+e.toString());
            }

            System.out .println("S-a citit: ");
            for(int i=0;i<buf.length ;i++)
                System.out .print((char)buf[i]);
            try{
        FileOutputStream iesire=new FileOutputStream ("RandomAccessFile_01.txt");
                iesire.write(buf);
                System.out .println("\nDatele au fost scrise in fisier.");
                iesire.close();
            }
            catch(Exception ex) {
                System.out .println("Eroare: "+ex.toString());
            }
            try{
RandomAccessFile         fisierRandom         =         new         RandomAccessFile
("RandomAccessFile_01.txt","rw");
                int lread;
                while((lread=fisierRandom.read(bufRead))!=-1){
                    temp=new String (bufRead,0,lread);
                    if(conv==0)temp=temp.toUpperCase();
                    else temp=temp.toLowerCase();
                    bufWrite=temp.getBytes();

                    fisierRandom.seek(fisierRandom.getFilePointer()-lread);
                    fisierRandom.write(bufWrite);
               }
               fisierRandom.close();
          }
          catch(IOException e) {
               System.out .println(e.getMessage());
          }
          try{
      FileInputStream intrare=new FileInputStream ("RandomAccessFile_01.txt");
               intrare.read(buf);
               intrare.close();
          }
          catch(IOException ex) {
               System.out .println("Eroare: "+ex.toString());
          }
      System.out .println("\nContinutul final al fisierului: ");
          for(int i=0;i<buf.length ;i++)
               System.out .print((char)buf[i]);
          }
}
```

Individual work

Complexity *

1. Using a KB reading mechanism (BufferedReader/InputStreamReader) input: a message of String type, a *day* as an integer, a *month* as a String and a *year* as an integer variable. The process will end by passing to a new line, or by typing a special String. Separate and display the tokens on different rows. Display all fields extracted from the stream as appeared.
Recommendation: use the StreamTokenizer class, the attributes sval, nval and the TT_EOL constant.
Consider the case in which the application is not aware of the entered data type (numbers, words). Use the constants TT_NUMBER, TT_WORD.

2. Implement the previous problem using a file as input source.

3. Read from the keyboard some 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 display messages if the year is leap. The program exits when the user types in X or x from KB. You may use *DateFormatSymbols* class for month conversion.

4. You are given a binary file which contains a sequence of characters representing a private Bitcoin 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 applied upon the private and public keys. Write the result in another binary file.

Complexity * *

5. Write a Java application which reads a set of text files that contain students data (CSV files). The files are stored on the local machine under the names Year_Y_Group_XXXX.txt . Agregate the information in these files using a *SequenceInputStream* and generate a new file which contains all the students ordered alphabetically.

6. 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).

7. Write a Java application which enables the serialization and deserialization of objects that represent arrays of *int* values. Populate an object with keyboard entered data, order the values and store the object in a file. Read the file and display the reconstructed array of values.

Complexity ***

8. 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 */ .

9. Read from a text file a grayscale image represented as a matrix of integer values. The image is followed by multiple convolution filters (*https://en.wikipedia.org/wiki/Kernel_(image_processing)*). Apply these filters to the original image and display both the original image, and the filtered results.