FLUXURI DE INTRARE/IEȘIRE

Operațiile de intrare/ieșire sunt realizate, în general, cu ajutorul claselor din pachetul `java.io`, folosind conceptul de *flux* (stream).

Un *flux* reprezintă o modalitate de transfer al unor informații în format binar de la o sursă către o destinație.

În funcție de modalitatea de prelucrare a informației, precum și a direcției canalului de comunicație, fluxurile se pot clasifica astfel:

- *după direcția canalului de comunicație*:
  - de intrare
  - de ieșire
- *după modul de operare asupra datelor:*
  - la nivel de octet (flux pe 8 biți)
  - la nivel de caracter (flux pe 16 biți)
- *după modul în care acționează asupra datelor:*
  - primitive (doar operațiile de citire/scriere)
  - procesare (adaugă la cele primitive operații suplimentare: procesare la nivel de buffer, serializare etc.)

În concluzie, pentru a deschide orice flux se instanțiază o clasă dedicată, care poate conține mai mulți constructori:

- un constructor cu un argument prin care se specifică calea fișierului sub forma unui șir de caractere;
- un constructor care primește ca argument un obiect de tip File;
- un constructor care primește ca argument un alt flux.

Clasa `File` permite operații specifice fișierelor și directoarelor, precum creare, ștergere, mutare etc., mai puțin operații de citire/scriere.

Metode uzuale ale clasei `File`:

- `String getAbsolutePath()` – returnează calea absolută a unui fișier;
- `String getName()` – returnează numele unui fișier;
- `boolean createNewFile()` – creează un nou fișier, iar dacă fișierul există deja metoda returnează `false`;
- `File[] listFiles()` – returnează un tablou de obiecte `File` asociate fișierelor dintr-un director.

Fluxurile primitive permit doar operații de intrare/ieșire. După modul de operarea asupra datelor, fluxurile primitive se împart în două categorii:

1. **prelucrare la nivel de caracter (fișiere text)**: informația este reprezentată prin caractere Unicode, aranjate pe linii (separatorul poate fi '`\r\n`' (Windows), '`\n`' (Unix/Linux) sau '`\r`' (Mac)).

   Informația fiind reprezentată prin caracter Unicode, se obține un flux pe 16 biți.

   Pentru deschiderea unui flux primitiv la nivel de caracter de intrare se instanțiază clasa `FileReader`, fie printr-un constructor care primește ca argument calea fișierului sub forma unui șir de caractere, fie printr-un constructor care primește ca argument un obiect de tip `File`.

```
FileReader fin = new FileReader("exemplu.txt");
```

sau

```
File f = new File("exemplu.txt");
FileReader fin = new FileReader(f);
```

Operația de citire a unui caracter se realizează prin metoda `int read()`.

Observație: Deschiderea unui fișier impune tratarea excepției `FileNotFoundException`.

Pentru deschiderea unui flux primitiv de ieșire la nivel de caracter se instanțiază clasa `FileWriter`, fie printr-un constructor care primește ca argument calea fișierului sub forma unui String, fie printr-un constructor care primește ca argument un obiect de tip `File`.

```
FileWriter fout = new FileWriter("exemplu.txt");
```

sau

```
File f = new File("exemplu.txt");
FileWriter fout = new FileWriter (f);
```

Pentru deschiderea unui flux primitiv de ieșire la nivel de caracter în modul `append` (adăugare la sfârșitul fișierului), se utilizează constructorul `FileWriter(String fileName, boolean append)`.

Dacă parametrul `append` are valoarea `true`, atunci operațiile de scriere se vor efectua la sfârșitul fișierului (dacă fișierul nu există, mai întâi se va crea un fișier vid). Dacă parametrul `append` are valoarea `false`, atunci operațiile de scriere se vor efectua la începutul fișierului (indiferent de faptul că fișierul există sau nu, mai întâi se va crea un fișier vid, posibil prin suprascrierea unuia existent).

Operația de scriere a unui caracter se realizează prin metoda `void write(int ch)`.
Clasa `FileWriter` pune la dispoziție și alte metode pentru a scrie informația într-un fișier text:

- `public void write(String string)` - scrie în fișier șirul de caractere transmis ca parametru
- `public void write(char[] chars)` - scrie în fișier tabloul de caractere transmis ca parametru

Observație: Scrierea informației într-un fișier impune tratarea excepției `IOException`.

Exemplu: Copierea caracter cu caracter a fișierului text `test.txt` în fișierul text `copie_caractere.txt`

```
FileReader fin = new FileReader("test.txt");
FileWriter fout = new FileWriter("copie_caractere.txt", true);

int c;
while((c = fin.read()) != -1)
   fout.write(c);
```

2. **prelucrare la nivel de octet(fișiere binare):** informația este reprezentată sub forma unui șir octeți neformatați (2 octeți nu mai reprezintă un caracter) și nu mai există o semnificație specială pentru caracterele '\r' și '\n'.

   Fișierele binare sunt des utilizate, deoarece acestea permit memorarea unor informații complexe, folosind un șablon, precum imagini, fișiere video etc. De exemplu, un fișier Word are un șablon specific, diferit de cel al unui fișier PDF.

   Pentru deschiderea unui flux primitiv de intrare la nivel de octet se instanțiază clasa `FileInputStream`, fie printr-un constructor care primește ca argument calea fișierului sub forma unui String, fie printr-un constructor care primește ca argument un obiect de tip File:

```
FileInputStream fin = new FileInputStream("test.txt");
```

   sau

```
File f = new File("exemplu.txt");
FileInputStream fin = new FileInputStream(f);
```

   Operația de citire a unui octet se realizează prin metoda `int read()`.
   Clasa `FileInputStream` pune la dispoziție și alte metode pentru a realiza citirea informației dintr-un fișier binar, precum:

   - `int read(byte[] bytes)` - citește un tablou de octeți și returnează numărul octeților citiți

   Pentru deschiderea unui flux primitiv de ieșire la nivel de octet se instanțiază clasa `OutputFileStream`, fie printr-un constructor care primește ca argument calea fișierului sub forma unui șir de caractere, fie printr-un constructor care primește ca argument un obiect de tip `File`:

```
FileOutputStream fout = new FileOutputStream("test.txt");
```

   sau

```
File f = new File("exemplu.txt");
FileOutputStream fout = new FileOutputStream(f);
```

   Operația de scriere a unui octet se realizează prin metoda `void write(int b)`.
   Clasa `FileOutputStream` pune la dispoziție și alte metode pentru a realiza scrierea informației într-un fișier binar:

   - `void write(byte[] bytes)` - scrie un tablou de octeți

   Exemple:

   1. Copierea directă a întregului conținut al fișierului text `test.txt` în fișierul text `copie_octeti.txt`.

```
FileInputStream fin = new FileInputStream("test.txt");
FileOutputStream fout = new FileOutputStream("copie_octeti.txt")
int dimFisier = fin.available();    //metoda returnează numărul de octeți din fișier
byte []buffer = new byte[dimFisier];
fin.read(buffer);          //se citesc toți octeții din fișierul de intrare
fout.write(buffer);        // se scriu toți octeții în fișierul de ieșire
```

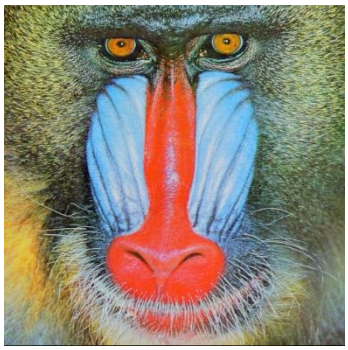
2. Formatul BMP (bitmap) pe 24 de biți este un format de fișier binar folosit pentru a stoca imagini color digitale bidimensionale având lățime, înălțime și rezoluție arbitrare. Practic, imaginea este considerată ca fiind un tablou bidimensional de pixeli, iar fiecare pixel este codificat prin 3 octeți corespunzători intensităților celor 3 canale de culoare R (red), G(green) și B(blue). Intensitatea fiecărui canal de culoare R, G sau B este dată de un număr natural cuprins între 0 și 255. De exemplu, un pixel cu valorile (0, 0 , 0) reprezintă un pixel de culoare neagră, iar un pixel cu valorile (255, 255, 255) unul de culoare albă.

Formatul BMP cuprinde o zonă cu dimensiune fixă, numita *header*, și o zonă de date cu dimensiune variabilă care conține pixelii imaginii propriu-zise. Header-ul, care ocupă primii 54 de octeți ai fișierului, conține informații despre formatul BMP, precum și informații despre dimensiunea imaginii, numărul de octeți utilizați pentru reprezentarea unui pixel etc. Dimensiunea imaginii în octeți este specificată în header printr-o valoare întreagă, deci memorată pe 4 octeți, începând cu octetul cu numărul de ordine 2. Dimensiunea imaginii în pixeli este exprimată sub forma $W \times H$, unde W reprezintă numărul de pixeli pe lățime, iar H reprezintă numărul de pixeli pe înălțime. Lățimea imaginii exprimată în pixeli este memorată pe patru octeți începând cu octetul al 18-lea din header, iar înălțimea este memorată pe următorii 4 octeți fără semn, respectiv începând cu octetul al 22- lea din header.

După cei 54 de octeți ai header-ului, într-un fișier BMP urmează zona de date, unde sunt memorate ÎN ORDINE INVERSĂ liniile de pixeli ai imaginii, deci ultima linie de pixeli din imagine va fi memorată prima, penultima linie va fi memorată a doua, …, prima linie din imagine va fi memorată ultima. Deoarece codarea unei imagini BMP pe 24 de biți într-un fișier binar respectă standardul *little-endian*, octeții corespunzători celor 3 canale de culoare RGB sunt memorați de la dreapta la stânga, în ordinea BGR!

Pentru rapiditatea procesării imaginilor la citire și scriere, imaginile în format BMP au proprietatea că fiecare linie este memorată folosind un număr de octeți multiplu de 4. Dacă este necesar, acest lucru de realizează prin adăugarea unor octeți de completare (*padding*) la sfârșitul fiecărei linii, astfel încât numărul total de octeți de pe fiecare linie să devină multiplu de 4. Numărul de octeți corespunzători unui linii este $3 \times W$ (câte 3 octeți pentru fiecare pixel de pe o linie). Astfel, dacă o imagine are $W = 11$ pixeli în lățime, atunci numărul de octeți de padding este 3 ($3 \times 11 = 33$ octeți pe o linie, deci se vor adăuga la sfârșitul fiecărei linii câte 3 octeți de completare, astfel încât să avem $33 + 3 = 36$ multiplu de 4 octeți). De obicei, octeții de completare au valoarea 0.

În continuare, considerăm imaginea *baboon.bmp* ca fiind imaginea de intrare, iar imaginea de ieșire ca fiind complementara sa, care se obține prin scăderea valorii fiecărui canal de culoare al unui pixel din valoarea 255 (valoarea maximă posibilă pe un canal de culoare).



baboon.bmp

complementara_baboon.bmp

Pentru a construi imaginea de ieșire, copiem mai întâi header-ul imaginii de intrare în imaginea de ieșire și apoi parcurgem fișierul de intrare la nivel de octet (variabila octet) pentru a accesa valorile de pe fiecare canal de culoare R, G și B din fiecare pixel și scriem în fișierul de ieșire valoarea complementară a octetului, respectiv `255 – octet`:

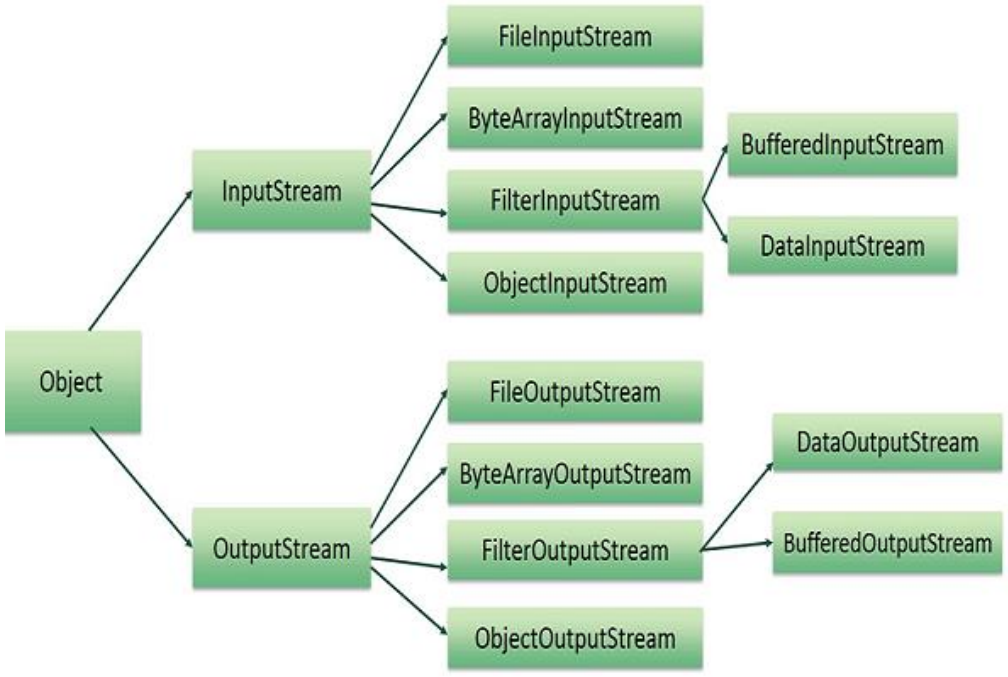
```
public class Prelucrare_BMP {
    public static void main(String[] args) throws FileNotFoundException,
                                                              IOException {
       FileInputStream fin = new FileInputStream("baboon.bmp");
       FileOutputStream fout = new FileOutputStream("complement_baboon.bmp");

       byte[] header = new byte[54];
       fin.read(header);
       fout.write(header);

       int octet;
       while((octet = fin.read()) != -1)
           fout.write(255 - octet);

       fin.close();
       fout.close();
    }
}
```

Fluxuri de procesare

Limbajul Java pune la dispoziție o serie de fluxuri de intrare/ieșire care au o structură stratificată pentru a adăuga funcționalități suplimentare pentru fluxurile primitive, întru-un mod dinamic și transparent. De exemplu, se poate adăuga la un flux primitiv binar de intrare operații care permit citirea tipurilor primitive (de exemplu, pentru a citi un număr întreg se grupează câte 4 octeți) sau a unui șir de caractere.

Această modalitate de a oferi implementări stratificate este cunoscută sub numele de *Decorator Pattern*. Conceptul în sine impune ca obiectele care adaugă funcționalități (*wrappers*) unui obiect să aibă o interfață comună cu acesta. În felul acesta, se obține transparența, adică un obiect poate fi folosit fie în forma primitivă, fie în forma superioară stratificată (decorat).

Limbajul Java pune la dispoziție o ierarhie complexă de clase pentru a prelucra fluxurile de procesare, așa cum se poate observa în figura de mai jos.



Observație: O ierarhie asemănătoare există și pentru fluxurile care procesează alte fluxuri la nivel de caracter.

Constructorii claselor pentru fluxurile de procesare nu primesc ca argument un dispozitiv extern de memorare a datelor, ci o referință a unui flux primitiv.

```
FluxPrimitiv flux  = new FluxPrimitiv(<lista arg>);
FluxDeProcesare fluxProcesare = new FluxDeProcesare(flux);
```

Exemple de fluxuri de procesare:

1. *Fluxurile de procesare* `DataInputStream/DataOutputStream`

Fluxul procesat nu mai este interpretat la nivel de octet, ci octeții sunt grupați astfel încât aceștia să reprezinte date primitive sau șiruri de caractere (`String`). Cele două clase furnizează metode pentru citirea și scrierea datelor la nivel de tip primitiv, prezentate in tabelul de mai jos:

DataInputStream	**DataOutputStream**
`boolean readBoolean()`	`void writeBoolean(boolean v)`
`byte readByte()`	`void writeByte(byte v)`
`char readChar()`	`void writeChar(int v)`
`double readDouble()`	`void writeDouble(double v)`
`float readFloat()`	`void writeFloat(float v)`
`int readInt()`	`void writeInt(int v)`
`long readLong()`	`void writeLong(long v)`
`short readShort()`	`void writeShort(int v)`
`String readUTF()`	`void writeUTF(String str)`

În exemplul de mai jos se realizează scrierea formatată a unui tablou de numere reale în fișierul binar *numere.bin*. Ulterior, folosind un flux binar se realizează citirea formatată a tabloului.

```
public class Fluxuri_date_primitive {
    public static void main(String[] args) {
      try(DataOutputStream fout = new DataOutputStream(
                                    new FileOutputStream("numere.bin"));) {
            double v[] = {1.5 , 2.6 , 3.7 , 4.8 , 5.9};
            fout.writeInt(v.length);
            for(int i = 0; i < v.length; i++)
                  fout.writeDouble(v[i]);
      }
      catch (IOException ex) {
            System.out.println("Eroare la scrierea in fisier!");
      }
         ..............................
      try(DataInputStream fin = new DataInputStream(
                                   new FileInputStream("numere.bin"));) {
            int n = fin.readInt();
            double []v = new double[n];

            for(int i = 0; i < v.length; i++)
                  v[i] = fin.readDouble();
            for(int i = 0; i < v.length; i++)
                  System.out.print(v[i] + " ");
      }
      catch (IOException ex) {
            System.out.println("Eroare la citirea din fisier!");
      }
    }
}
```

2. *Fluxuri de procesare pentru citirea/scrierea datelor folosind un buffer*

Operațiile de citire/scriere la nivel de caracter/octet, specifice fluxurilor primitive, conduc la un număr mare de accesări ale fluxului respectiv (și, implicit, ale dispozitivului de memorie externă pe care este stocat fișierul asociat), ceea ce poate afecta eficiența din punct de vedere al timpului de executare. În scopul de a elimina acest neajuns, fluxurile de procesare la nivel de buffer introduc în procesele de scriere/citire o zonă auxiliară de memorie, astfel încât informația să fie accesată în blocuri de caractere/octeți având o dimensiune predefinită, ceea ce conduce la scăderea numărului de accesări ale fluxului respectiv.

Clase pentru citirea/scrierea cu buffer:

- `BufferedReader, BufferedWriter` – fluxuri de procesare la nivel de buffer de caractere
- `BufferedInputStream, BufferedOutputStream` – fluxuri de procesare la nivel de buffer de octeți

Constructori:

- `FluxProcesareBuffer flux = new FluxProcesareBuffer(new FluxPrimitiv("cale fișier"));`
- `FluxProcesareBuffer flux = new FluxProcesareBuffer(new FluxPrimitiv("cale fișier"), int dimBuffer);`

Dimensiunea implicită a buffer-ului utilizat este de 512 octeți.

Metodele uzuale ale acestor clase sunt: `read/readline`, `write`, `flush` (golește explicit buffer-ul, chiar dacă acesta nu este plin).

Exemplu: Fișierul *date.in* conține un text dispus pe mai multe linii. În fișierul *date.out* sunt afișate, pe fiecare linie, cuvintele sortate crescător lexicografic.

```
public class CitireBuffer {
    public static void main(String[] args) {
      try(BufferedReader fin = new BufferedReader(new FileReader("date.in"));
        BufferedWriter fout = new BufferedWriter(new FileWriter("date.out"));)
        {
            String linie;

            while((linie=fin.readLine())!=null)
              {
                String cuv[] = linie.split(" ");

                Arrays.sort(cuv);
                System.out.println(Arrays.toString(cuv));
                for(int i=0; i<cuv.length; i++)
                    fout.write(cuv[i]+" ");
                fout.write("\n");
              }
        }
        catch (FileNotFoundException ex) {
            System.out.println("Fisierul nu exista!");
        }
        catch(IOException ex) {
              System.out.println("Operatie de citire/scriere esuata!");
        }

    }
}
```

Fluxuri de procesare cu acces aleatoriu

Toate fluxurile de procesare prezentate anterior sunt limitate la o accesare secvențială a sursei/destinației de date. Astfel, nu putem accesa (citi/scrie) direct un anumit octet/caracter/valoare din flux, ci trebuie să accesăm, pe rând, toate valorile aflate înaintea sa, de la începutul fluxului respectiv. Dacă pentru unele categorii de fluxuri accesarea secvențială este indispensabilă (de exemplu, în cazul unor fluxuri cu ajutorul cărora se transmit date într-o rețea), în cazul anumitor tipuri de fișiere se poate opta pentru o accesare directă, mai eficientă în cazul în care nu este necesară procesarea tuturor datelor din fișier, ci doar a unora a căror poziție este cunoscută (de exemplu, lățimea unei imagini în format *bitmap* (BMP) este memorată pe 4 octeți, începând cu octetul 18, iar pe următorii 4 octeți, începând cu octetul 22, este memorată înălțimea sa).

Pentru accesarea aleatorie a octeților unui fișier, în limbajul Java este utilizată clasa `RandomAccessFile`, care nu aparține niciunei ierarhii de clase menționate până acum. Accesarea aleatorie a octeților unui fișier se realizează prin intermediul unui *cursor* asociat fișierului respectiv (file pointer) care memorează numărul de

ordine al octetului curent (în momentul deschiderii unui fișier, cursorul asociat este poziționat pe primul octet din fișier – octetul cu numărul de ordine 0). Practic, fișierul este privit ca un tablou unidimensional de octeți memorat pe un suport extern, iar cursorul reprezintă indexul octetului curent. Orice operație de citire/scriere se va efectua asupra octetului curent, după care se va actualiza valoarea cursorului. De exemplu, dacă octetul curent este octetul 10 și vom scrie în fișier valoarea unei variabile de tip `int`, care se memorează pe 4 octeți, valoarea cursorului va deveni 14.

Deschiderea unui fișier cu acces aleatoriu se poate realiza utilizând unul dintre cei 2 constructori ai clasei `RandomAccessFile`, unul având ca parametru un obiect de tip File, iar celălalt având ca parametru calea fișierului sub forma unui șir de caractere:

- `RandomAccessFile(File file, String mode)`
- `RandomAccessFile(String name, String mode)`

Parametrul `mode` este utilizat pentru a indica modalitatea de deschidere a fișierului, astfel:
- `"r"` – fișierul este deschis doar pentru citire (dacă fișierul nu există, se va lansa excepția `FileNotFoundException`);
- `"rw"` – fișierul este deschis pentru citire și scriere (dacă fișierul nu există, se va crea unul vid).

Clasa `RandomAccessFile` implementează interfețele `DataInput` și `DataOutput` (care sunt implementate, de exemplu, și de clasele `DataInputStream`/`DataOutputStream`), deci conține metode pentru citirea/scrierea:
- *octeților sau tablourilor de octeți* – utilizând metode `read`/`write` asemănătoare celor din clasele `FileInputStream`/`FileOutputStream`;
- *valori de tip primitiv sau șiruri de caractere* – utilizând metodele `readTip`/`writeTip` asemănătoare celor din clasele `DataInputStream` și `DataOutputStream`

În cazul apariției unor erori la scrierea/citirea datelor se va lansa o excepție de tipul `IOException`.

În afara metodelor pentru citirea/scrierea datelor, clasa `RandomAccessFile` conține și metode specifice pentru poziționarea cursorului fișierului:

- `long getFilePointer()` – furnizează valoarea curentă a cursorului asociat fișierului, raportată la începutul fișierului (octetul cu numărul de ordine 0);
- `void seek(long pos)` – mută cursorul asociat fișierului pe octetul cu numărul de ordine `pos` față de începutul fișierului (octetul cu numărul de ordine 0);
- `int skipBytes(int n)` – mută cursorul asociat fișierului peste `n` octeți față de poziția curentă.

Observație: În limbajul Java, toate fișierele binare sunt considerate în mod implicit ca fiind de tip *big-endian* în mod implicit, respectiv octetul cel mai semnificativ dintr-un grup de octeți va fi memorat primul în fișierul binar. În cazul în care o aplicație Java manipulează fișiere binare de tip *little-endian* (octetul cel mai semnificativ dintr-un grup de octeți va fi memorat ultimul), create, de exemplu, utilizând limbajele C/C++ în sistemul de operare Microsoft Windows, acest fapt poate genera probleme foarte mari, deoarece datele vor fi interpretate eronat!

De exemplu, să considerăm o valoare `int x = 720`, care se memorează pe 4 octeți în limbajele C/C++/Java. În baza 2, valoarea `x` este egală cu `1011010000`, deci, folosind standardul *little-endian*, reprezentarea sa internă va fi egală cu `11010000 | 00000010 | 00000000 | 00000000` (prin `|` am delimitat octeții). Dacă

această reprezentare binară va fi interpretată folosind standardul *big-endian*, atunci ea va fi considerată ca fiind egală, în baza 10, cu `-805175296`!

Pentru a rezolva această problemă, se poate proceda în două moduri:

- dacă valoarea este de tip `char`, `short`, `int` sau `long`, se poate utiliza metoda `reverseBytes` din clasa înfășurătoare corespunzătoare. De exemplu, citim dintr-un fișier `fin` cu acces aleatoriu o valoare `int x=fin.readInt()` și schimbăm ordinea octeților `x=Integer.reverseBytes(x)` sau, direct, prin `x=Integer.reverseBytes(fin.readInt())`.

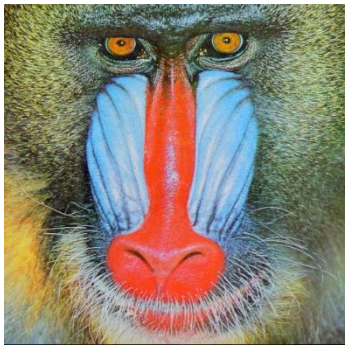
- o altă variantă, care poate fi utilizată pentru orice tip de date, constă în utilizarea unui obiect de tip `ByteBuffer` pentru manipularea șirurilor de octeți:

```
//citim din fișier o valoare de tip double direct,
//sub forma unui șir de 8 octeți
byte []valoareDouble = new byte[8];
fin.read(valoareDouble);

//alocăm un obiect de tip ByteBuffer care să permită manipularea
//a 8 octeți și stabilim ordinea lor ca fiind little-endian
ByteBuffer aux = ByteBuffer.allocate(8);
aux.order(ByteOrder.LITTLE_ENDIAN);

//încărcăm șirul de octeți în obiectul ByteBuffer și apoi
//preluăm valoarea de tip double astfel obținută
aux.put(valoareDouble);
double x = aux.getDouble();
```

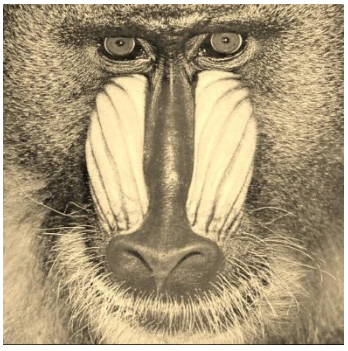
Exemplu: O imagine color poate fi transformată într-o imagine sepia înlocuind valorile (R, G, B) ale fiecărui pixel cu valorile (R', G', B') definite astfel:

$$R' = \min\{[0.393 * R + 0.769 * G + 0.189 * B], 255\}$$
$$G' = \min\{[0.349 * R + 0.686 * G + 0.168 * B], 255\}$$
$$B' = \min\{[0.272 * R + 0.534 * G + 0.131 * B], 255\}$$

unde prin $[x]$ am notat partea întreagă a numărului real x.



baboon.bmp (color)



baboon.bmp (sepia)

În următoarea aplicație Java, vom utiliza un fișier cu acces aleatoriu pentru a afișa dimensiunea imaginii în octeți și în pixeli, după care vom transforma imaginea color inițială într-una sepia, ținând cont de faptul că fișierele BMP sunt implicit de tip *little-endian*:

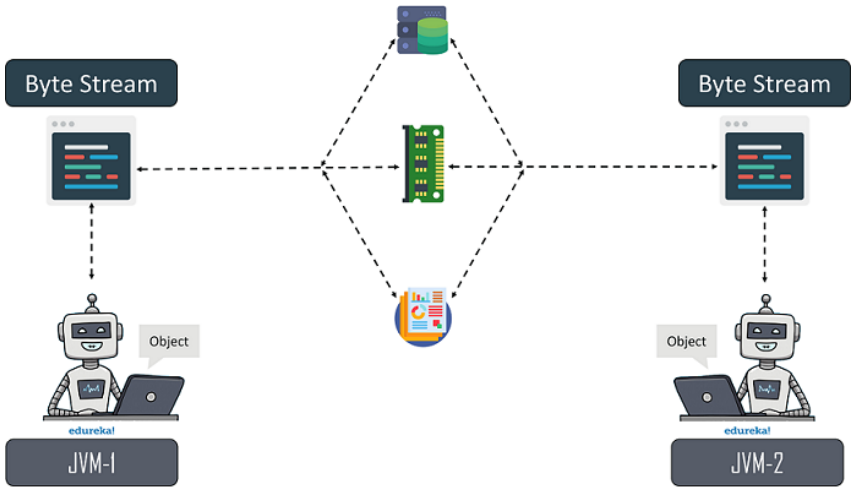
```
public class Prelucrare_BMP_sepia {
    public static void main(String[] args) throws FileNotFoundException,
                                                          IOException {

      //deschidem fișierul în mod mixt, deoarece trebuie să efectuăm și
      //operații de citire și operații de scriere
      RandomAccessFile img = new RandomAccessFile("baboon.bmp", "rw");

      //citim din fișier dimensiunea imaginii în octeți și o afișam
      byte []b = new byte[4];
      ByteBuffer aux = ByteBuffer.allocate(4);

      img.seek(2);
      img.read(b);

      aux.put(b);
      aux.order(ByteOrder.LITTLE_ENDIAN);
      int imgBytes = aux.getInt(0);

      System.out.println("Dimensiunea imaginii: " + imgBytes + " bytes");

      //citim din fișier dimensiunea imaginii în pixeli și o afișam
      img.seek(18);

      int imgWidth = img.readInt();
      imgWidth = Integer.reverseBytes(imgWidth);

      int imgHeight = img.readInt();
      imgHeight = Integer.reverseBytes(imgHeight);

      System.out.println("Dimensiunea imaginii: " + imgWidth + " x " +
                                                  imgHeight + " pixeli");

      //calculăm padding-ul imaginii și îl afișăm
      int imgPadding;

      if(imgWidth % 4 != 0)
            imgPadding = 4 - (3 * imgWidth) % 4;
      else
            imgPadding = 0;

      System.out.println("Padding-ul imaginii: " + imgPadding + " bytes");

      //modificăm imaginea color într-una sepia

      //în tabloul de octeți pixelRGB citim valorile pixelului curent color
      byte []pixelRGB = new byte[3];
```

```
      //în tabloul de octeți auxRGB vom calcula noile valori ale pixelului
      //curent transformat în sepia, folosind formulele de mai sus și ținând
      //cont de faptul că în fișier canalele de culoare sunt în ordinea BGR
      byte []auxRGB = new byte[3];
      double tmp = 0;

      //mutăm cursorul la începutul zonei de date, imediat după header-ul
      //de 54 de octeți
      img.seek(54);

      for(int h = 0; h < imgHeight; h++) {
        for(int w = 0; w < imgWidth; w++) {
            //citim valorile RGB ale pixelului curent în ordinea BGR
            img.read(pixelRGB);

            //calculăm valorile sepia ale pixelului curent
            tmp = 0.272*Byte.toUnsignedInt(pixelRGB[0]) +
                  0.534*Byte.toUnsignedInt(pixelRGB[1]) +
                  0.131*Byte.toUnsignedInt(pixelRGB[2]);
            auxRGB[0] = (byte) (tmp <= 255 ? tmp : 255);

            tmp = 0.349*Byte.toUnsignedInt(pixelRGB[0]) +
                  0.686*Byte.toUnsignedInt(pixelRGB[1]) +
                  0.168*Byte.toUnsignedInt(pixelRGB[2]);
            auxRGB[1] = (byte) (tmp <= 255 ? tmp : 255);

            tmp = 0.393*Byte.toUnsignedInt(pixelRGB[0]) +
                  0.769*Byte.toUnsignedInt(pixelRGB[1]) +
                  0.189*Byte.toUnsignedInt(pixelRGB[2]);
            auxRGB[2] = (byte) (tmp <= 255 ? tmp : 255);

            //ne întoarcem 3 octeți în fișier pentru a suprascrie valorile
            //color ale pixelului curent cu cele sepia calculate mai sus
            img.seek(img.getFilePointer() - 3);
            img.write(auxRGB);
        }

        //după ce am prelucrat toți pixelii de pe o linie, sărim peste
        //pixelii de padding
        img.skipBytes(imgPadding);
      }

      img.close();
    }
}
```

SERIALIZAREA OBIECTELOR

Ciclul de viață al unui obiect este determinat de executarea programului, respectiv obiectele instanțiate în cadrul său sunt stocate în memoria internă, astfel încât, după ce acesta își termină executarea, zona de memorie alocată programului este eliberată. În cadrul aplicațiilor, de cele mai multe ori, se dorește salvarea obiectelor între diferite rulări ale programului sau se dorește ca acestea să fie transmise printr-un un canal de comunicație. O soluție aparent simplă pentru rezolvarea acestei probleme ar constitui-o salvarea stării unui obiect (valorile datelor membre) într-un fișier (text sau binar) și restaurarea ulterioară a acestuia, pe baza valorilor salvate, folosind un constructor al clasei. Totuși, această soluție devine foarte complicată dacă unele date membre sunt referințe către alte obiecte, deoarece ar trebui salvate și restaurate și stările acelor obiecte externe! Mai mult, în acest caz nu s-ar salva funcționalitățile obiectului (metodele sale) și constructorii.

Limbajul Java permite o rezolvare simplă și eficientă a acestei probleme prin intermediul mecanismelor de *serializare* și *deserializare* (sursa imaginii: https://www.edureka.co/blog/serialization-in-java/):



Serializarea este mecanismul prin care un obiect este transformat într-o secvență de octeți din care acesta să poată fi refăcut ulterior, iar *deserializare*a reprezintă mecanismul invers serializării, respectiv dintr-o secvență de octeți serializați se restaurează obiectul original.

Utilizarea mecanismului de serializare prezintă mai multe avantaje:

- obiectele pot fi salvate/restaurate într-un mod unitar pe/de pe diverse medii de stocare (fișiere binare, baze de date etc.);
- obiectele pot fi transmise foarte simplu între mașini virtuale Java diferite, care pot rula pe calculatoare având arhitecturi sau sisteme de operare diferite;
- timpul necesar serializării sau deserializării unui obiect este mult mai mic decât timpul necesar salvării sau restaurării unui obiect pe baza valorilor datelor sale membre (de exemplu, în momentul deserializării unui obiect nu se apelează constructorul clasei respective);
- cea mai simplă și mai rapidă metodă de clonare a unui obiect (*deep copy*) o reprezintă serializare/deserializarea sa într-un/dintr-un tablou de octeți.

Obiectele unei **clase sunt serializabile dacă respectiva clasă implementează interfața `Serializable`**. Această interfață este una de marcaj, care nu conține nicio metodă abstractă, deci, prin implementarea sa clasa respectivă doar anunță mașina virtuală Java faptul că dorește să-i fie asigurat mecanismul de serializare. O clasă nu este implicit serializabilă, deoarece clasa `java.lang.Object` nu implementează interfața `Serializable`. Totuși, anumite clase standard, cum ar fi clasa `String`, clasele înfășurătoare (wrapper), clasa `Arrays` etc., implementează interfața `Serializable`.

Pentru prezentarea mecanismului de serializare/deserializare, vom considera definită clasa `Student`, care implementează interfața `Serializable`, având datele membre `String` nume, `int` grupa, un tablou `note` cu elemente de tip `int` pentru a reține notele unui student, `double` medie și o dată membră statică `facultate` de tip `String`, respectiv metodele de tip `set/get` corespunzătoare, metoda `toString()` și constructori:

```
public class Student implements Serializable
{
    private static String facultate;
    private String nume;
    private int grupa, note[];
    private double medie;
    ...
}
```

Serializarea unui obiect se realizează astfel:

- se deschide un flux binar de ieșire utilizând clasa `java.io.ObjectOutputStream`:
  ```
  FileOutputStream file = new FileOutputStream("studenti.bin");
  ObjectOutputStream fout = new ObjectOutputStream(file);
  ```

- se salvează/scrie obiectul în fișier apelând metoda `void writeObject(Object ob)`:
  ```
  Student s = new Student("Ion Popescu", 241, new int[]{10, 9, 10, 7, 8});
  fout.writeObject(s);
  ```

Deserializarea unui obiect se realizează astfel:

- se deschide un flux binar de intrare utilizând clasa `java.io.ObjectInputStream`:
  ```
  FileInputStream file = new FileInputStream("studenti.bin");
  ObjectInputStream fin = new ObjectInputStream(file);
  ```

- se citește/restaurează obiectul din fișier apelând metoda `Object readObject()`:
  ```
  Student s = (Student)fin.readObject();
  ```

Mecanismul de serializare a unui obiect presupune salvarea, în format binar, a următoarelor informații despre acesta:

- denumirea clasei de apartenență;
- versiunea clasei de apartenență, implicit aceasta fiind hash-code-ul acesteia, calculat de către mașina virtuală Java;
- valorile datelor membre de instanță.
- antetele metodelor membre.

Observații:

- Implicit NU se serializează datele membre statice și nici corpurile metodelor, ci doar antetele lor.
- Explicit NU se serializează datele membre marcate prin modificatorul `transient` (de exemplu, s-ar putea să nu dorim salvarea anumitor informații confidențiale: salariul unei persoane, parola unui utilizator etc.).
- Serializarea nu tine cont de specificatorii de acces, deci se vor serializa și datele/metodele private!
- În momentul sterilizării unui obiect se va serializa întregul graf de dependențe asociat obiectului respectiv, adică obiectul respectiv și toate obiectele referite direct sau indirect de el.

Exemplu:
Considerăm clasa `Nod` care modelează un nod al unei liste simplu înlănțuite:

```
class Nod implements Serializable
{
    Object data;
    Nod next;

    public Nod(Object data)
    {
        this.data = data;
        this.next = null;
    }
}
```

Folosind clasa `Nod`, vom construi o listă circulară, formată din numerele naturale cuprinse între 1 și 10, pe care apoi o vom salva/serializa în fișierul binar `lista.ser`, scriind în fișier doar primul său nod (obiectul `prim`) – restul nodurilor listei vor fi salvate/serializate automat, deoarece ele formează graful de dependențe asociat obiectului `prim`:

```
public class Serializare_listă_circulară
{
    public static void main(String[] args)
    {
        Nod prim = null, ultim = null;

        for (int i = 1; i <= 10; i++)
        {
            Nod aux = new Nod(i);

            if (prim == null) prim = ultim = aux;
            else
            {
                ultim.next = aux;
                ultim = aux;
            }
        }
        ultim.next = prim;

        System.out.println("Lista care va fi serializată:");
        Nod aux = prim;
        do
```

```
        {
            System.out.print(aux.data + " ");
            aux = aux.next;
        }
        while(aux != prim);

        try (ObjectOutputStream fout = new ObjectOutputStream(
                                        new FileOutputStream("lista.ser")))
        {
            fout.writeObject(prim);
        }
        catch (IOException ex) { System.out.println("Excepție: " + ex); }
    }
}
```

Pentru a restaura lista circulară salvată/serializată în fișierul binar `lista.ser`, vom citi/deserializa din fișier doar primul său nod (obiectul `prim`), iar restul nodurilor listei vor fi restaurate/deserializate automat, deoarece ele formează graful de dependențe asociat obiectului `prim` (evident, clasa `Nod` trebuie să fie vizibilă):

```
public class Deserializare_listă_ciculară
{
    public static void main(String[] args)
    {
        try (ObjectInputStream fin = new ObjectInputStream(
                                        new FileInputStream("lista.ser")))

        {
            Nod prim = (Nod) fin.readObject();



            System.out.println("Lista deserializata:");
            Nod aux = prim;
            do
            {
                System.out.print(aux.data + " ");
                aux = aux.next;
            }
            while(aux != prim);

            System.out.println();
        }
        catch (Exception ex)
        {
            System.out.println("Excepție: " + ex);
        }
    }
}
```

În exemplul prezentat, graful de dependențe asociat obiectului `prim` este unul ciclic, deoarece lista este circulară (deci există o referință indirectă a obiectului `prim` către el însuși), dar mecanismul de serializare gestionează fără probleme o astfel de situație complicată!

- Dacă un obiect care trebuie serializat conține referințe către obiecte neserializabile, atunci va fi generată o excepție de tipul `NotSerializableException`.
- Dacă o clasă serializabilă extinde o clasă neserializabilă, atunci datele membre accesibile ale superclasei nu vor fi serializate. În acest caz, superclasa trebuie să conțină un constructor fără argumente pentru a inițializa în procesul de restaurare a obiectului datele membre moștenite.
- Dacă se modifică structura clasei aflată pe mașina virtuală care realizează serializarea obiectelor (de exemplu, prin adăugarea unui câmp nou privat, care, oricum, nu va fi accesibil), fără a se realiza aceeași modificare și pe mașina virtuală destinație, atunci procesul de deserializare va lansa la executare excepția `InvalidClassException`. Excepția apare deoarece cele două clase nu mai au aceeași versiune. Practic, versiunea unei clase se calculează în mod implicit de către mașina virtuală Java printr-un algoritm de hash care este foarte sensibil la orice modificare a clasei.

  În practică, sunt diferite situații în care se dorește modificarea clasei pe mașina virtuală care realizează procesul de serializare (de exemplu, adăugând date sau metode private care vor fi utilizate doar intern), fără a afecta, însă, procesul de deserializare. O soluție în acest sens o constituie introducerea unei noi date membre în clasă, `private static final long serialVersionUID`, prin care se definește explicit versiunea clasei - caz în care mașina virtuală Java nu va mai calcula, implicit, versiunea clasei respective pe baza structurii sale. Un exemplu bun în acest sens se găsește în pagina https://www.baeldung.com/java-serial-version-uid.