EXCEPȚII

O **excepție** este un eveniment care întrerupe executarea normală a unui program. Exemple de excepții: împărțirea unui număr întreg la 0, încercarea de deschidere a unui fișier inexistent, accesarea unui element inexistent într-un tablou, procesarea unor date de intrare incorecte etc.

Tratarea excepțiilor devine stringentă în aplicații complexe, formate din mai multe module (de exemplu, o interfață grafică care implică apelurile unor metode din alte clase). De regulă, rularea unui program presupune o succesiune de apeluri de metode, spre exemplu, metoda `main()` apelează metoda `f()` a unui obiect, aceasta la rândul său apelează o metodă `g()` a altui obiect ș.a.m.d. astfel încât, în orice moment, există mai multe metode care și-au început executarea, dar nu și-au încheiat-o deoarece punctul de executare se află într-o altă metodă. Succesiunea de apeluri de metode a căror executare a început, dar nu s-a și încheiat este numită **call-stack** (stiva cu apeluri de metode) și reprezintă un concept important în logica tratării erorilor.

Să presupunem, de exemplu, că avem o aplicație cu o interfață grafică care conține un buton "`Statistică persoane`". În momentul apăsării butonului, se apelează o metodă "`AcțiuneButon`", pentru a trata evenimentul, care la rândul său apelează o metodă "`CalculStatistică`" dintr-o altă clasă, iar aceasta, la rândul său, apelează o metodă "`ÎncărcareDateDinFișier`". Se obține astfel un call-stack. În această situație, pot să apară mai multe excepții care pot proveni din diferite metode aflate pe call-stack: calea fișierului cu datele persoanelor este greșită sau fișierul nu există, unele persoane au datele eronate în fișier etc. Indiferent de metoda în care va apărea o excepție, aceasta trebuie semnalată utilizatorului în interfața grafică, adică trebuie **să aibă loc o propagare a excepției, fără a bloca funcționalitatea aplicației.**

O variantă de rezolvare ar fi utilizarea unor coduri pentru excepții, dar acest lucru ar complica foarte mult codul (multe `if`-uri), iar coduri precum -1, -20 etc. nu sunt descriptive pentru excepția apărută. În limbajul Java, există un mecanism eficient de tratare a excepțiilor. Practic, o excepție este un obiect care încapsulează detalii despre excepția respectivă, precum metoda în care a apărut, metodele din call-stack afectate, o descriere a sa etc.

Tipuri de excepții:

- **erori:** sunt generate de hardware sau de JVM, ci nu de program, ceea ce înseamnă că nu pot fi anticipate, deci *nu este obligatorie tratarea lor* (exemplu: `OutOfMemoryError`)
- **excepții la compilare**: sunt generate de program, ceea ce înseamnă că pot fi anticipate, deci *este obligatorie tratarea lor* (exemple: `IOException`, `SQLException` etc.)
- **excepții la rulare**: sunt generate de o situație particulară care poate să apară la rulare, ceea ce înseamnă că pot fi foarte numeroase (nu există o listă completă a lor), *deci nu este obligatorie tratarea lor* (exemple: `IndexOutOfBoundsException`, `NullPointerException`, `ArithmeticException` etc.)

Deoarece există mai multe situații în care pot apărea excepții, Java pune la dispoziție o ierarhie complexă de clase dedicate (Fig. 1).

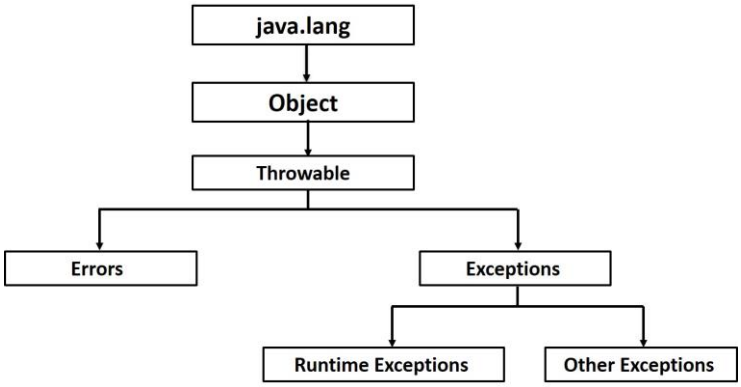


Fig. 1 - Ierarhia de clase pentru tratarea excepțiilor

Se poate observa cum există o multitudine de tipuri derivate din `Exception` sau `RuntimeException`, distribuite în diverse pachete Java. De regulă, excepțiile nu grupate într-un singur pachet (nu există un pachet `java.exception`), ci sunt definite în aceleași pachete cu clasele care le generează. De exemplu, `IOException` este definită în `java.io`, `AWTException` în `java.awt` etc. Lista de excepții definite în fiecare pachet poate fi găsită în documentația Java API.

Exemple de excepții uzuale:

- `IOException` - apare în operațiile de intrare/ieșire (de exemplu, citirea dintr-un fișier sau din rețea). O subclasă a clasei `IOException` este `FileNotFoundException`, generată în cazul încercării de deschidere a unui fișier inexistent;
- `NullPointerException` - folosirea unei referințe cu valoarea `null` pentru accesarea unui membru public sau default dintr-o clasă;
- `ArrayIndexOutOfBoundsException` - folosirea unui index incorect, respectiv negativ sau strict mai mare decât dimensiunea fizică a unui tablou - 1;
- `ArithmeticException` - operații aritmetice nepermise, precum împărțirea unui număr întreg la 0;
- `IllegalArgumentException` – utilizarea incorectă a unui argument pentru o metodă. O subclasă a clasei `IllegalArgumentException` este `NumberFormatException` care corespunde erorilor de conversie a unui `String` într-un tip de date primitiv din cadrul metodelor `parseTipPrimitiv` ale claselor wrapper;
- `ClassCastException` - apare la conversia unei referințe către un alt tip de date incompatibil;
- `SQLException` - excepții care apar la interogarea serverelor de baze de date.

Mecanismul folosit pentru manipularea excepțiilor predefinite este următorul:

- *generarea excepției*: când apare o excepție, JVM instanțiază un obiect al clasei Exception care încapsulează informații despre excepția apărută;
- *lansarea/aruncarea excepției*: obiectul generat este transmis mașinii virtuale;
- *propagarea excepției*: JVM parcurge în sens invers call-stack-ul, căutând un handler (un cod) care tratează acel tip de eroare;
- *prinderea și tratarea excepției*: primul handler găsit în call-stack este executat ca reacție la apariția erorii, iar dacă nu se găsește niciun handler, atunci JVM oprește executarea programului și afișează un mesaj descriptiv de eroare.

Sintaxa utilizată pentru tratarea excepțiilor:

```
try {
        bloc de instrucțiuni
}
catch(Excepție_A e) {
   Tratare excepție A
}
catch(Excepție_B e) {
   Tratare excepție B (mai generală)
}
finally {
    Bloc care se execută întotdeauna
}
```

Observații:

- Un bloc `try-catch` poate să conțină mai multe blocuri `catch`, însă acestea trebuie să fie specificate de la particular către general (și în această ordine vor fi și tratate). De exemplu `Excepție_A` este o subclasă a clasei `Excepție_B`

Exemplu: Următoarea aplicație, care citește două numere întregi dintr-un fișier text, conține un bloc `catch` pentru a trata excepția care poate să apară dacă se încercă deschiderea unui fișier inexistent, dar poate să conțină și un blocuri `catch` care tratează excepții de tipul `ArithmeticException` și/sau excepții de tipul `InputMismatchException`.

```
public class Test {
    public static void main(String[] args) {
        int a, b;
        try {
            Scanner f = new Scanner(new File("numere.txt"));
            a = f.nextInt();
            b = f.nextInt();
            double r;
            r = a / b;
            System.out.println(r);
        }
        catch(FileNotFoundException e) {
            System.out.println("Fisier inexistent");
        }
        catch(InputMismatchException e) {
            System.out.println("Format incorect al unui numar");
        }
        catch(ArithmeticException e) {
            System.out.println("Impartire la 0");
        }
        finally {
            System.out.println("Bloc finally");
        }
    }
}
```

- Blocurile `catch` se exclud reciproc, respectiv o excepție nu poate fi tratată de mai multe blocuri `catch`.

  Exemplu:
  - dacă nu există fișierul `numere.txt`, atunci se lansează și se tratează doar excepția `FileNotFoundException`, afișând-se în fereastra System mesajul `"Fisier inexistent"`, fără a se mai executa și blocurile `ArithmeticException` și `InputMismatchException`;
  - dacă în fișierul `numere.txt` sunt valorile `abc 0`, atunci se lansează și se tratează doar `InputMismatchException`, fără a se executa și blocul `ArithmeticException`;
  - dacă în fișierul `numere.txt` sunt valorile `13 0`, atunci se lansează și se tratează `ArithmeticException`, fără a se mai executa `InputMismatchException`.

- Blocul `finally` nu are parametri și poate să lipsească, dar, dacă există, atunci se execută întotdeauna, indiferent dacă a apărut o excepție sau nu. Scopul său principal este acela de a eliberarea anumite resurse deschise, de exemplu, fișiere sau conexiuni de rețea.

- Blocul `finally` va fi executat întotdeauna după blocurile `try` și `catch`, astfel:
  - dacă în blocul `try` nu apare nicio excepție, atunci blocul `finally` este executat imediat după `try`;
  - dacă în blocul `try` este aruncată o excepție, atunci:
    - dacă exista un bloc `catch` corespunzător, acesta va fi executat după întreruperea executării blocului `try`, urmat de blocul `finally`;
    - dacă nu exisă un bloc `catch`, atunci se execută blocul `finally` imediat dupa blocul `try`, după care JVM caută un handler în metoda anterioară din call-stack;
  - blocul `finally` se execută chiar și atunci când folosim instrucțiunea `return` în cadrul blocurilor `try` sau `catch`!

  Exemplu: După rularea programului de mai jos, se vor afișa mesajele `Înainte de return` și `Bloc finally`!

```
public class Test {
   static void test() {
        try {
             System.out.println("Înainte de return");
             return;
        }
        finally {
             System.out.println("Bloc finally");
        }
   }

   public static void main(String[] args) {
         test();
   }
}
```

Observație: instrucțiunea `try-catch` este un dispecer de excepții, similar instrucțiunii `switch(TipExceptie)`, direcționând-se astfel fiecare excepție către blocul de cod care o tratează.

Excepții definite de către programator

Așa cum am precizat mai sus, standardul Java oferă o ierarhie complexă de clase pentru manipularea diferitelor tipuri de excepții, care pot să acopere multe dintre erorile întâlnite în programare. Totuși, pot exista situații în care trebuie sa fie tratate anumite excepții specifice pentru logica aplicației (de exemplu, excepția dată de adăugarea unui element într-o stivă plină, introducerea unui CNP invalid, utilizarea unei date calendaristice anterioare unui proces etc.). În plus, excepțiile standard deja existente nu descriu întotdeauna detaliat o situație de eroare (de exemplu, `IllegalArgumentException` poate fi o informație prea vagă, în timp ce `CNPInvalidException` descrie mai bine o eroare și poate să permită o tratare separată a sa).

În acest sens, programatorul își poate defini propriile excepții, prin clase care extind fie clasa `Exception` (o excepție care trebuie să fie tratată), fie clasa `RuntimeException` (o excepție care nu trebuie să fie tratată neapărat).

Lansarea unei excepții se realizează prin clauza `throw new ExcepțieNouă(<listă argumente>)`.

Exemplu: Vom implementa o stivă de numere întregi folosind un tablou unidimensional, precum și excepții specifice, astfel:

- definim o clasă `StackException` pentru manipularea excepțiilor specifice unei stive:

```
public class StackException extends Exception {
    public StackException(String mesaj) {
        super(mesaj);
    }
}
```

- definim o interfață `Stack` în care precizăm operațiile specifice unei stive, inclusiv excepțiile:

```
public interface Stack {
    void push(Object item) throws StackException;
    Object pop() throws StackException;
    Object peek() throws StackException;
    boolean isEmpty();
    boolean isFull();
    void print() throws StackException;
}
```

- definim o clasă `StackArray` în care implementăm operațiile definite în interfața `Stack` utilizând un tablou unidimensional, iar posibilele excepții le lansăm utilizând excepții descriptive de tipul `StackException`:

```
public class StackArray implements Stack {
    private Object[] stiva;
    private int varf;

    public StackArray(int nrMaximElemente) {
        stiva = new Object[nrMaximElemente];
        varf = -1;
    }
```

```
    @Override
    public void push(Object x) throws StackException {
        if (isFull())
            throw new StackException("Nu pot să adaug un element într-o
                                                        stivă plină!");

        stiva[++varf] = x;
    }

    @Override
    public Object pop() throws StackException {
        if (isEmpty())
            throw new StackException("Nu pot să extrag un element dintr-o
                                                         stivă vidă!");

        Object aux = stiva[varf];
        stiva[varf--] = null;
        return aux;
    }

    @Override
    public Object peek() throws StackException {
        if (isEmpty())
            throw new StackException("Nu pot să accesez elementul din
                                            vârful unei stive vide!");

        return stiva[varf];
    }

    @Override
    public boolean isEmpty() {
        return varf == -1;
    }

    @Override
    public boolean isFull() {
        return varf == stiva.length - 1 ;
    }

    @Override
    public void print() throws StackException {
        if (isEmpty())
            throw new StackException("Nu pot să afișez o stivă vidă!");

        System.out.println("Stiva: ");
        for(int i = varf; i >= 0; i--)
            System.out.print(stiva[i] + " ");
        System.out.println();
    }
}
```

- Testăm clasa `StackArray` efectuând operații de tip `push` și `pop` în mod aleatoriu asupra unei stive care poate să conțină maxim 3 numere întregi:

```
public class Test_StackArray {
    public static void main(String[] args) {
        StackArray st = new StackArray(3);

        Random rnd = new Random();
        for(int i = 0; i < 20; i++)
            try {
                int aux = rnd.nextInt();

                if(aux % 2 == 0)
                    st.push(1 + rnd.nextInt(100));
                else
                    st.pop();

                st.print();
            }
            catch(StackException ex) {
                System.out.println(ex.getMessage());
            }
    }
}
```

„Aruncarea" unei excepții

Dacă în corpul unei metode nu se tratează o anumită excepție sau un set de excepții, în antetul metodei se poate folosi clauza **`throws`** pentru ca acesta/acestea să fie tratate de către o metodă apelantă.

Sintaxa:

```
tip_returnat numeMetoda(<listă argumente>) throws listaExcepții
```

Exemplu:

```
void citire() throws IOException {
   System.in.read();
}

void citeșteLinie(){
   citire();
}
```

Metoda `citeșteLinie`, la rândul său, poate să "arunce" excepția `IOException` sau să o trateze printr-un bloc `try-catch`.

În concluzie, aruncarea unei excepții de către o metodă presupune, de fapt, pasarea explicită a responsabilității către codul apelant al acesteia. Vom proceda astfel numai când dorim să forțam codul client să trateze excepția în cauză.

Observație: La redefinirea unei metode care "aruncă" excepții, nu se pot preciza prin clauza `throws` excepții suplimentare.

Observație: Începând cu Java 7, a fost introdusă instrucțiunea *try-with-resources* care permite închiderea automată a unei resurse, adică a unui surse de date de tip flux (de exemplu, un flux asociat unui fișier, o conexiune cu o bază de date etc.) .

Sintaxă:

```
try(deschidere Resursă_1; Resursă_2) {
      .........
}
catch(...) {
      .........
}
```

Pentru a putea fi utilizată folosind o instrucțiune de tipul *try-with-resources*, clasa corespunzătoare unei resurse trebuie să implementeze interfața `AutoCloseable`. Astfel, în momentul terminării executării instrucțiunii se va închide automat resursa respectivă. Practic, după executarea instrucțiunii *try-with-resources* se vor apela automat metodele `close` ale resurselor deschise.

Exemplu: Indiferent de tipul lor, fluxurile asociate fișierelor se închid folosind metoda `void close()`, de obicei în blocul `finally` asociat instrucțiunii `try-catch` în cadrul căreia a fost deschis fluxul respectiv:

```
try {
      FileOutputStream fout = new FileOutputStream("numere.bin");
      DataOutputStream dout = new DataOutputStream(fout);
      .........
}
catch (...) {
      .........
}
finally {
      if(dout != null)
            dout.close();
}
```

Toate tipurile de fluxuri bazate pe fișiere implementează interfața `AutoCloseable`, deci pot fi deschise utilizând o instrucțiune de tipul *try-with-resources*.

```
try(FileOutputStream fout = new FileOutputStream("numere.bin");
    DataOutputStream dout = new DataOutputStream(fout);)) {
      .........
}
catch (...) {
      .........
}
```

Observație: În momentul închiderii unui flux stratificat, așa cum este fluxul `dout` din exemplul de mai sus, JVM va închide automat și fluxul primitiv pe bază căruia acesta a fost deschis!

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 `InputFileStream`, 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
   ```

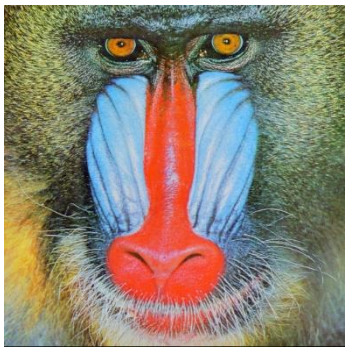
2. Formatul BMP (bitmap) pe 24 de biți este un format de fișier binar folosit pentru a stoca imagini color digitale bidimensionale având lățime, înălțime și rezoluție arbitrare. Practic, imaginea este considerată ca fiind un tablou bidimensional de pixeli, iar fiecare pixel este codificat prin 3 octeți 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`:

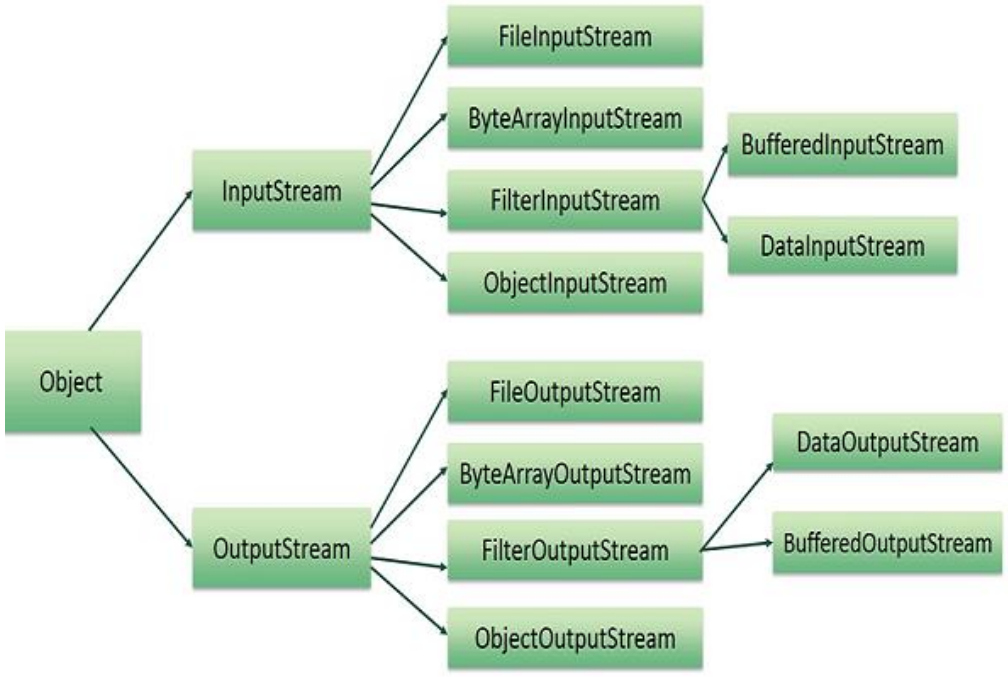
```
public class Prelucrare_BMP {
    public static void main(String[] args) throws FileNotFoundException,
                                                   IOException {
       FileInputStream fin = new FileInputStream("baboon.bmp");
       FileOutputStream fout = new FileOutputStream("complement_baboon.bmp");

       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())`.

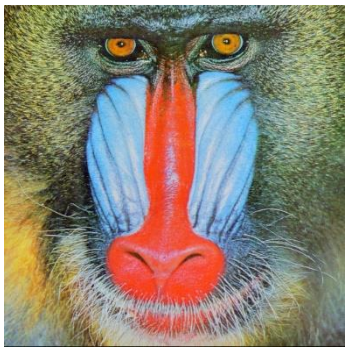
- o altă variantă, care poate fi utilizată pentru orice tip de date, constă în utilizarea unui obiect de tip `ByteBuffer` pentru manipularea șirurilor de octeți:

```
//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();
```

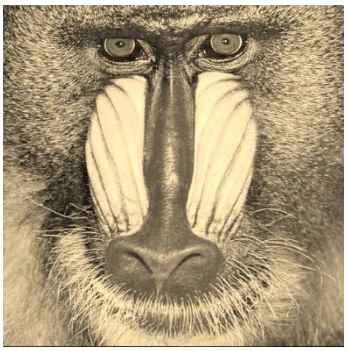
Exemplu: O imagine color poate fi transformată într-o imagine sepia înlocuind valorile (R, G, B) ale fiecărui pixel cu valorile (R', G', B') definite astfel:

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

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();
    }
}
```