Lab. 7: Stream-uri de intrare și de ieșire.

Serializare. Deserializare.

Cuprins

Recapitulare laborator 6

Laboratorul 6 a fost dedicat tratării excepţiilor şi colecţiilor. În ceea ce priveşte excepţiile în Java, este util să ne amintim cele mai bune practici de lucru cu acestea, de exemplu folosirea unor excepţii specifice, prinderea lor atunci când pot fi tratate corespunzător sau chiar folosirea unor excepţii proprii, cu nume care să respecte convenţiile.

Colecţiile în Java sunt obiecte ce grupează mai multe obiecte într-o singură unitate. Amintim principalele tipuri de colecţii: liste, mulţimi matematice, tabele de dispersie. Acestea sunt tipuri de date *parametrizate,* deci trebuie să precizăm tipul de date care formează colecţia. De asemenea, pentru a parcurge elementele unei colecţii se folosesc iteratori sau o construcţie `for` specială, *for-each.*

 Încercaţi să răspundeţi la următoarele întrebări:

- Ce sunt excepţiile în Java? Care este clasa părinte pentru orice excepţie ?
- Care sunt cele mai bune practici de lucru cu excepţii ?
- Ce tipuri de colecţii cunoaşteţi? Daţi exemple.
- Cum se poate realiza ştergerea unui element dintr-o colecţie ?
- Când se foloseşte un ArrayList şi când un LinkedList ? Care sunt complexităţile principalelor metode din celor două clase ?
- Când se foloseşte un HashMap şi când se foloseşte un HashSet ?

Fluxuri de intrare şi de ieşire

Referinţă: https://docs.oracle.com/javase/tutorial/essential/io/index.html

Introducere

Operaţiile de intrare-ieşire (I/O) sunt realizate, în general, cu ajutorul claselor din pachetul `java.io`. Facilităţile de intrare/ieşire din Java au la bază noţiunea de **flux**. Un flux este un obiect asociat unei succesiuni de elemente (octeți sau caractere), citite şi scrise secvenţial. Pentru un flux de intrare sursa datelor poate fi un fişier, dar şi un şir sau tablou de octeți, respectiv caractere. Pentru un flux de ieşire, datele transmise sunt stocate într-un fişier sau într-un tablou de octeți, respectiv caractere.

Clasele Java pentru I/O sunt separate în funcţie de tipul operaţiei (de citire sau scriere) şi există două în funcţie de tipul datelor pe care operează:

- fluxuri de octeți. Asociate adesea cu fişiere binare. Sunt clasele derivate din InputStream şi OutputStream.

- fluxuri de caractere. Asociate adesea cu fişiere text. Sunt clasele derivate din Reader şi Writer.

Atenţie! Majoritatea metodelor din aceste clase aruncă excepţii de tip IOException sau derivate din aceasta, de exemplu FileNotFoundException. Acestea trebuie prinse!

Observaţie: Atunci când lucrăm cu stream-uri, avem următorul algoritm:

```
try {
  deschide stream
  while(mai exista date) {
      citeşte / scrie date;
  }
} finally {
  închide stream;
}
```

Fluxuri la nivel de octet

Structura de clase şi interfeţe cel mai des utilizate:

```
Object
   DataInput (interfață)
   InputStream (clasă abstractă)
     FileInputStream
     FilterInputStream
        DataInputStream implements DataInput
   DataOutput (interfață)
   OuputStream (clasă abstractă)
     FileOutputStream
     FilterOutputStream
        DataOutputStream implements DataOutput
```

Clasa InputStream

- este superclasa tuturor claselor pentru lucrul cu fluxuri de intrare la nivel de octet;
- este o clasă abstractă, iar toate clasele care o extind trebuie să implementeze o metodă care să intoarcă următorul octet al inputului;
- principalele metode puse la dispoziţie sunt următoarele:
  - `abstract int read()` – citeşte şi întoarce un octet, iar, in caz de EOF (*end of stream*), întoarce -1 (se semnalează astfel sfârşitul de flux).
  - `int read(byte[] b)` – citeşte un număr de maxim `b.length` octeţi din stream şi îi stochează în b. Întoarce numărul de octeţi citiţi sau, în caz de EOF, -1.

- `int read(byte[] b, int from, int len)` - citeşte un număr de maxim `len` octeţi din stream şi îi stochează în b, începând de la poziţia `from`. Întoarce numărul de octeţi citiţi sau, în caz de EOF, -1.
- `int available()` – întoarce numărul maxim de octeţi care pot fi citiţi din acest stream fără ca operaţia să se blocheze (folositoare în cazul pipe-urilor, conexiunilor Internet etc).
- `int skip(long n)` - sare peste n octeţi, întorcând numărul efectiv de octeţi peste care s-a sarit, sau -1 în caz de EOF.
- `void close()` - închide streamul şi eliberează orice resursă asociată cu acesta

Clasa OutputStream

- este superclasa tuturor claselor pentru lucrul cu fluxuri de ieşire la nivel de octet;
- un stream de output acceptă octeţi şi îi trimite la o destinaţie
- metodele suportate sunt:
  - `abstract void write(int b)`
  - `void write(byte[]b)`
  - `void write(byte[] b, int off, int len)`
  - `void flush()`
  - `void close()`
- operaţiile de `write` sunt complementare celor de `read` din InputStream.

Exemplu 1: Scrierea şi citirea de octeţi. Dorim să copiem un fişier cu un nume dat într-un alt fişier dat.

```
public class FileDuplicator {

   public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        FileInputStream fis = null;
        FileOutputStream fos = null;

        try {
            System.out.println("File to copy: ");
            String fileName = sc.next();
            fis = new FileInputStream(fileName);

            System.out.println("Destination file: ");
            fileName = sc.next();
            fos = new FileOutputStream(fileName);

            int c;
```

```
            while ((c = fis.read()) != -1) {
                fos.write(c);
            }

            System.out.println("Done!");
        } catch (FileNotFoundException exc) {
            System.out.println("File not found, exiting...");
        } catch (IOException exc) {
            System.out.println("IOException occurred: ..." +
                               exc.getMessage();
        } finally {
            if(fis != null) {
                fis.close();
            }
            if(fos != null) {
                fos.close();
            }
        }
    }
}
```

Exemplul 2: Scrierea şi citirea de date primitive. Într-o primă etapă vom citi de la intrarea standard un număr natural *n* şi apoi *n* numere reale; vom crea în directorul curent un fişier cu numele *out.dat* în care vom scrie datele citite. Într-o a doua etapă vom citi din fişierul *out.dat* valoarea *n* şi cele *n* numere şi le vom tipări la ieşirea standard.

```
public class DOSExample {

   public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);
        System.out.print("n = ");
        int n = sc.nextInt();
        DataOutputStream dos = null;

        try {
            dos = new DataOutputStream(new
FileOutputStream("out.dat"));
            dos.writeInt(n);
            System.out.println("The n numbers are: ");
            for(int i = 0; i < n; ++i) {
                dos.writeDouble(sc.nextDouble());
```

```
            }
        } catch (IOException exc) {
            System.out.println("IOException  occurred: " +
exc.getMessage());
        } finally {
            try {
                if(dos != null) {
                    dos.close();
                }
            } catch (IOException e) {
                System.out.println("Error closing the file...");
            }

        }
    }
}
```

Observaţii:

- prin `new FileOutputStream("out.dat")` este creat fişierul `out.dat`
- fluxul `dos` de tipul `DataOutputStream` foloseşte metodele `writeInt(int v)` şi `writeDouble(double v)` (anunţate în interfaţa `DataOutput` şi implementate în clasa `DataOutputStream`) pentru a scrie în fişierul `out.dat`.
- pentru închiderea fişierului este chemată metoda `close()` în blocul `finally`, astfel încât resursa să se închidă indiferent dacă operaţiile s-au încheiat cu succes sau nu.

```
public class DISExample {

   public static void main(String[] args) {
        DataInputStream dis = null;

        try {
            dis = new DataInputStream(new
FileInputStream("out.dat"));
            int n = dis.readInt();
            System.out.println("n = " + n);
            for(int i = 0; i < n; ++i) {
                System.out.println(dis.readDouble() + "\t");
            }
            System.out.println();
```

```
        } catch (IOException exc) {
            System.out.println("IOException occurred: "+
exc.getMessage());
        } finally {
            try {
                if(dis != null) {
                    dis.close();
                }
            } catch (IOException e) {
                System.out.println("Error closing the file..");
            }
        }
    }
}
```

Fluxuri ce lucrează la nivel de caracter

Structura de clase şi interfeţe cel mai des utilizate:

```
Object
  Writer (clasă abstractă)
     OutputStreamWriter
        FileWriter
     PrintWriter
  Reader (clasă abstractă)
     InputStreamReader
        FileReader
```

În vârful ierarhiilor claselor care lucrează cu caractere se află `Reader` şi `Writer`. Acestea oferă funcţionalităţi asemănătoare cu cele din InputStream / OutputStream, cu diferenţa că este folosit caracterul şi nu octetul ca unitate de informaţie. Dacă dorim să scriem / citim şiruri de caractere, trebuie să folosim `Reader` şi `Writer`.

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

        PrintWriter out = null;
        try {
            out = new PrintWriter("in.txt");
            out.println(10.0);
            out.println(5);
            out.println("Hello");
        } catch(IOException e) {
```

```
            e.printStackTrace();
        } finally {
            if (out != null) {
                out.close();
            }
        }
    }
}
```

Fluxuri Buffered

Este un fapt cunoscut că operatiile I/O sunt mari consumatoare de timp. În forma lor de bază, clasele I/O **nu** folosesc buffer-e, în sensul că, de exemplu, fiecare operaţie `write` are drept urmare o operaţie **fizică** de `write`, pentru un stream fişier. Ar fi mult mai convenabil să adunam mai multe date într-un buffer şi să le scriem pe toate deodată.

Această funcţionalitate este oferită de clasele BufferedInputStream / BufferedOutputStream pentru lucrul cu octeţi şi de BufferedReader şi BufferedWriter pentru lucrul cu caractere.

Exemplu: Dorim să citim linie cu linie un fişier şi să afişăm liniile la ieşirea standard.

```
public class BuffExample {

   public static void main(String[] args) {
        try {
            BufferedReader br = new BufferedReader(new
FileReader("inputs/in.txt"));
            String line;
            while((line = br.readLine()) != null) {
                System.out.println(line);
            }
        } catch (FileNotFoundException e) {
            System.out.println(e.getMessage());
        } catch (IOException e) {
            e.printStackTrace();
        }

    }
}
```

Fluxuri punte octet - caracter

Exista **conversii** de la şiruri de octeţi la şiruri de caractere. Filtrul ce realizează această conversie este InputStreamReader, care obţine perspectiva **caracter** asupra unui stream **octet**. `InputStreamReader` foloseste o codificare dată ca parametru în constructor sau foloseşte codificarea implicită.

O aplicaţie foarte importantă a acestui mecanism o reprezintă citirea de la **consolă**, ţinând cont de faptul că `System.in` are tipul `InputStream`.

Exemplu: Dorim să scriem într-un fişier liniile pe care le citim de la consolă.

```
public class BuffExample {

   public static void main(String[] args) {
        FileWriter fileWriter = null;
        try {
            BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
            fileWriter = new FileWriter("out.txt");
            String line;
            while((line = br.readLine()) != null) {
                if(line.equals("")) {
                    break;
                }
                fileWriter.write(line + "\n");
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if(fileWriter != null) {
                try {
                    fileWriter.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

Serializare şi deserializare

În multe situații apare necesitatea ca unul sau mai multe obiecte să poată fi memorate pentru a fi folosite şi după încheierea aplicației care le-a creat. Pentru aceasta starea lor trebuie salvată într-un mod care să permită refacerea lor ulterioară. Java oferă mecanismele de *serializare* şi *deserializare*, care realizează tocmai salvarea, respectiv refacerea obiectelor. Una dintre aplicațiile evidente ale acestor mecanisme constă în serializarea obiectelor pe un sistem, urmată de transmiterea lor la distanță şi deserializarea lor pe o nouă maşină gazdă pentru a fi folosite în continuare.

Serializarea constă în memorarea, într-un anumit format, a stării obiectelor într-un tablou de octeți sau într-un fişier. Acestea au un caracter serial: pot accepta numai secvențe de octeți. Facilitatea ca obiectele să poată "supraviețui" şi după teminarea executării programului care le-a creat se mai numeşte *persistența datelor*.

Pentru ca un obiect să poată fi serializat, el trebuie să implementeze interfața Serializable. Serializarea se realizează prin intermediul fluxurilor:

- `DataInputStream / DataOutputStream`
- `ObjectInputStream / ObjectOutputStream`

Exemplu: Considerăm clasa `Angajat` - o clasă ale cărei obiecte dorim să le serializăm.

```
public class Angajat implements Serializable {
    String nume;
    int varsta;
    int salariu;
    static String firma;

    public Angajat(String n, int v, int s) {
        nume = n;
        varsta = v;
        salariu = s;
    }

    public void print() {
        System.out.println("Firma:\t\t" + firma + "\n" +
                "Nume:\t\t" + nume + "\n" +
                "Varsta:\t\t" + varsta + "\n" +
                "Salariul:\t" + salariu + "\n");
    }
}
```

Clasa `Serial` conţine o metodă main în care se serializează doi Angajaţi într-un fişier.

```
public class Serial {
   public static void main(String[] sir) {
       Angajat.firma = "SRL Serial";
       Angajat angajat1 = new Angajat("Vasile", 25, 1485);
       Angajat angajat2 = new Angajat("Ion", 24, 420);
       FileOutputStream fos = null;
       ObjectOutputStream oos = null;

       try {
           fos = new FileOutputStream("Serial");
           oos = new ObjectOutputStream(fos);
           oos.writeObject(angajat1);
           oos.writeObject(angajat2);
       } catch (FileNotFoundException e) {
           e.printStackTrace();
       } catch (IOException e) {
           e.printStackTrace();
       } finally {
           if(oos != null) {
               try {
                   oos.close();
               } catch (IOException e) {
                   e.printStackTrace();
               }
           }

           if(fos != null) {
               try {
                   fos.close();
               } catch (IOException e) {
                   e.printStackTrace();
               }
           }
       }
   }
}
```

Observaţii:

- Clasa `ObjectOutputStream` extinde clasa abstractă `OutputStream`.

- Prin invocarea metodei `writeObject` se realizează serializarea (într-un mod standard) a obiectelor create şi scrierea rezultatului serializării în fluxul de ieşire.
- Clasa `Angajat` implementează interfața `Serializable`. Prin abuz de limbaj vom numi *clasă serializabilă* o clasă ce implementează această interfață. ***Numai instanțele unei clase serializabile pot fi serializate*!**

În continuare, realizăm deserializarea obiectelor salvate anterior.

```
public class Serial1 {
   public static void main(String args[]) {
        FileInputStream fis = null;
        ObjectInputStream ois = null;
        try {
            fis = new FileInputStream("Serial");
            ois = new ObjectInputStream(fis);
            Angajat angajat1 = (Angajat) ois.readObject();
            Angajat angajat2 = (Angajat) ois.readObject();
            angajat1.print();
            angajat2.print();

        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } finally {
            if (ois != null) {
                try {
                    ois.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }

            if (fis != null) {
                try {
                    fis.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
```

```
}
```

Observaţii:

- Clasa `ObjectInputStream` extinde clasa abstractă `InputStream`.
- Prin invocarea metodei `readObject` se realizează deserializarea (într-un mod standard) a obiectelor din fluxul de intrare.
- Metoda `readObject` întoarce un obiect de tipul `Object`, ceea ce face necesară conversia explicită la tipul `Angajat`.

Important! La serializare se transmit numai valorile câmpurilor de instanţă ale obiectului, nu şi cele statice, care aparţin claselor, nu şi instanţelor. De asemenea, la serializarea unui obiect se serializează *întregul graf asociat obiectului respectiv* (*graful asociat unui obiect* constă din obiectul respectiv, dar şi din obiectele referite direct sau indirect de el). Această facilitate este foarte utilă şi poate fi folosită de exemplu pentru:

- transmiterea unui arbore; este suficient să serializăm rădăcina şi vor fi serializate toate vârfurile arborelui, împreună cu structura de arbore;
- transmiterea unei liste circulare: este suficient să serializăm primul element al listei şi vor fi serializate toate elementele listei, împreună cu structura de listă circulară; în particular, serializarea "nu se încurcă" dacă un obiect din graful asociat se autoreferă.

Observaţie: Există posibilitatea să se salveze şi câmpurile statice, însă acestea trebuie incluse **explicit** în fluxul de ieşire. De exemplu, pentru exemplul de mai sus:

```
oos.writeUTF(Angajat.firma);
Angajat.firma = ois.readUTF();
```

Exerciţii

1. Se dă un fişier `in.txt`. Să se sorteze liniile acestuia şi să se scrie în fişierul `out.txt`.
2. Se dă un fişier cu mai multe numere separate prin spaţiu. Citiţi de la tastatură un număr *x* şi apoi afisaţi din *x* în *x* numerele din fişier.
3. (*) Dat un anumit director de pe disc, parcurgeţi recursiv toţi copii acestuia şi creaţi câte un fişier în fiecare director care să conţină numele tutoror fişierelor de pe acel nivel. Hint: pentru manipularea fişierelor consultaţi şi clasele File şi Files.
4. Se consideră clasa `Student (nume, prenume, medie)` şi clasa `Grupă(identificator, listaStudenti)`. Să se serializeze două obiecte de tip `Grupă` într-un fişier. Să se deserializeze apoi şi să afişeze fiecare obiect.
5. Să se serializeze o listă circulară într-un fişier. Să se deserializeze conţinutul fişierului şi să se afişeze lista.