Séance 5

Entrées-sorties : Flux, Fichiers, Sérialisation, Réseaux

Dans cette séance, la gestion des entrées-sorties en Java est abordée. La manipulation des fichiers regroupe l'ensemble des concepts de base pour gérer les entrées-sorties. Une fois celle-ci acquise, il est aisé d'établir des communications réseaux, d'exploiter la sérialisation pour créer des objets persistants, de transmettre ces objets persistants via le réseau ...

Sommaire

5.1 Flux en entrée/sortie . . . 51
5.1.1 Traitement des erreurs d'entrée/sortie . . . 51
5.1.2 Les flux d'octets . . . 51
5.1.3 Les flux de caractères . . . 52
5.1.4 Flux avec tampons . . . 53
5.2 Fichiers, réseau et sérialisation (objets persistants) . . . 53
5.2.1 Fichiers . . . 53
5.2.2 Réseau . . . 56
5.2.3 Sérialisation . . . 58

Le code de départ pour illustrer les notions de cours est contenu dans l'archive `io.zip`, qui peut être importée directement dans `eclipse`.

5.1 Flux en entrée/sortie

Les informations présentées dans cette partie sont reprises de ce tutoriel d'Oracle : http://docs.oracle.com/javase/tutorial/essential/io/index.html

Un flux est une suites séquentielle de données. On distingue les flux en entrée qui permettent à une application de lire les données une par une, des flux en sortie qui permettent d'envoyer des données, toujours une par une. Les flux en entrée et en sortie sont modélisés respectivement par les interfaces `InputStream` et `OutputStream`.

5.1.1 Traitement des erreurs d'entrée/sortie

Le traitement des erreurs lors de la lecture ou de l'écriture de données se fait naturellement via le mécanisme des exceptions. La classe `IOException` est la classe mère de toutes les erreurs d'entrée/sortie. Il suffit donc de la gérer pour prendre en charge n'importe quel type d'exception liée aux entrées/sorties.

On peut affiner le traitement d'une erreur en capturant une des exceptions *filles* de `IOEXception`, dont voici les principales :
— `CharacterCodingException` (encodage du fichier non-conforme)
— `EOFException` (fin du fichier atteinte)
— `FileNotFoundException` (fichier manquant)
— `InterruptedIOException` (communication interrompue)
Le schéma global d'un code réalisant des entrées/sorties est dont le suivant :

```
try {
... des entrees / sorties ...
} catch (IOException ioe) {
        System.err.println("Erreur lors de la communcation : "+ioe);
}
```

5.1.2 Les flux d'octets

Les flux d'octets sont les flux "bruts", i.e. les flux de plus bas niveau, offrant le moins de fonctionnalités mais susceptibles de transporter tout type de données. Les flux de données plus "évolués" exploitent de manière implicite ces flux d'octets.

Il existe plusieurs classes pour les flux d'octets, qui fonctionnent toutes de manière similaire. Dans le cas de flux basé sur des fichiers, on utilise les classes `FileInputStream` et `FileOutputStream`. La seconde partie de ce chapitre décrit de manière détaillée la manipulation des fichiers.

Voici un exemple de code, qui recopie les données du fichier `xanadu.txt`, octet par octet.

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

        FileInputStream in = null;
        FileOutputStream out = null;

        try {
            in = new FileInputStream("xanadu.txt");
            out = new FileOutputStream("outagain.txt");
            int c;

            while ((c = in.read()) != -1) {
                out.write(c);
            }
           if (in != null)
               in.close();
           if (out != null)
               out.close();
        }

        catch (IOException ioe) {
                System.err.println("Erreur lors de la communcation : "+ioe);
        }
    }
}
```

Notez que la valeur -1 est renvoyée par la méthode `read` lorsqu'il n'y a plus de données à lire.

Il est de plus important de toujours les libérer les ressources liées aux flux en invoquant la méthode `close`.

Exercice 64. *(⋆) Testez la classe `CopyBytes`.*

Les flux d'octets ne sont pas recommandés car ils sont trop rudimentaires. Dans notre exemple, le fichier est un fichier texte et peut donc être vu comme un flux de caractères, pour lesquels il existe des flux plus adaptés que nous décrivons dans la sous-section suivante.

5.1.3 Les flux de caractères

Le stockage des caractères en interne est effectué avec le codage unicode, qui est universel. Java effectue les traductions nécessaires lorsque les codages utilisés par l'émetteur et le récepteur diffèrent.

Les flux de caractères sont modélisés par les interfaces `Reader` et `Writer`. Dans le cas des fichiers, les classes correspondantes sont `FileReader` et `FileWriter`.

Avec celles-ci, notre exemple précédent s'écrit :

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

        FileReader inputStream = null;
        FileWriter outputStream = null;

        try {
            inputStream = new FileReader("xanadu.txt");
            outputStream = new FileWriter("characteroutput.txt");

            int c;
            while ((c = inputStream.read()) != -1) {
                outputStream.write(c);
            }
            if (inputStream != null)
              inputStream.close();
            if (outputStream  != null)
              outputStream .close();
         }
         catch (IOException ioe) {
                System.err.println("Erreur lors de la communcation : "+ioe);
         }
    }
}
```

Exercice 65. *(⋆) Testez la classe `CopyCharacters` en utilisant différents encodages pour le fichier en entrée.*

5.1.4 Flux avec tampons

La lecture ou l'écriture de données une par une pose des problèmes de performance : il est plus judicieux d'envoyer ou de récupérer des données par blocs.

Pour cela, on utilise des flux avec tampon, via les classes `BufferedReader` ou `BufferedWriter`. On instancie ces classes à partir de flux "classiques" :

```
inputStream = new BufferedReader(new FileReader("xanadu.txt"));
outputStream = new BufferedWriter(new FileWriter("characteroutput.txt"));
```

Dans le cas de flux en écriture avec tampons, il faut veiller à vider le tampon via la méthode `flush()` pour s'assurer que les données ont bien été toutes envoyées avant de fermer le flux.

Pour gérer les fichiers textes, on peut lire une ligne en entier via la méthode `readLine` et utiliser la classe `PrintWriter` pour écrire des données ligne par ligne.

Notre exemple précédent s'écrit alors :

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

        BufferedReader inputStream = null;
        PrintWriter outputStream = null;

        try {
            inputStream = new BufferedReader(new FileReader("xanadu.txt"));
            outputStream = new PrintWriter(new FileWriter("characteroutput.txt"));

            String l;
            while ((l = inputStream.readLine()) != null) {
                outputStream.println(l);
            }
           if (inputStream != null)
             inputStream.close();
           if (outputStream  != null)
            outputStream .close();
         }
         catch (IOException ioe) {
               System.err.println("Erreur lors de la communcation : "+ioe);
         }
     }
}
```

Exercice 66. *(⋆) Testez la classe `CopyLines`.*

5.2 Fichiers, réseau et sérialisation (objets persistants)

Dans cette partie, nous décrivons tout d'abord comment lire et écrire dans des fichiers. Les communications via un réseau sont ensuite traitées ainsi que le mécanisme de sérialisation des objets (i.e. encodage de l'état d'un objet dans le but de pouvoir le transmettre).

5.2.1 Fichiers

Java7 vient d'introduire une nouvelle bibliothèque (NIO2) pour la gestion des systèmes de fichiers. Nous allons nous baser sur cette dernière, car elle offre des fonctionnalités plus complètes que les versions antérieures. Les informations présentées dans cette partie sont issues du document "Java 7 : découverte de NIO2 pour l'accès aux fichiers" par Alexis Hassier : http://www.jtips.info/index.php?title=Java7/NIO2-FileSystem.

Avant Java 7, la classe centrale pour les accès aux fichier était la classe `File` de la bibliothèque `java.io`. En Java 7, cette classe existe toujours mais ses fonctionnalités sont accessibles via de nouvelles classes de la bibliothèque `java.nio.file`.

Manipulation de chemins L'interface `Path` permet de représenter un chemin dans le système de fichiers.

Voici un exemple d'utilisation de cette interface :

```
Path p = Paths.get("/home/gothmog/workspace/JavaCours");
Path homePath = Paths.get(System.getProperty("user.home")); // homePath = repertoire personnel
    (=/home/gothmog)
Path cheminRelatif = homePath.relativize(p); // chemin relatif vers p a partir de homePath
System.out.println(cheminRelatif); // workspace/JavaCours
Path repertoireCourant = Paths.get("").toAbsolutePath(); // chemin complet vers la position
   actuelle
System.out.println(repertoireCourant); // /home/gothmog/workspace/JavaCours
```

Il est aisé de passer à l'ancienne classe `File` à partir de `Path` et vice-versa :

```
File myFile = myPath.toFile(); // de Path vers File
Path myPath = myFile.toPath(); // de File vers Path
```

Lecture du contenu d'un fichier La lecture du contenu d'un fichier est réalisable avec un objet implémentant l'interface `BufferedReader` (le tampon permet de prendre en charge des fichiers volumineux).

Exemple de code pour lire et afficher le contenu d'un fichier texte utilisant l'encodage `UTF8` :

```
Charset UTF8 = Charset.forName("UTF-8");
Path codeMain =  Paths.get("./src/scolarite/Main.java");

try {
        BufferedReader br = Files.newBufferedReader(codeMain,UTF8);
        String ligne = br.readLine();
        while (ligne!=null){
                System.out.println(ligne);
                ligne=br.readLine();
        }
        br.close();
}
catch (IOException ioe) {
        System.err.println("Erreur de lecture : "+ioe);
}
```

Exercice 67. *(⋆) Tester ce code. Que se passe-t-il si on donne un chemin incorrect lors de l'initialisation de `codeMain` ?*

Pour les petits fichiers, la classe `Files` permet de récupérer directement tout le contenu d'un fichier via les méthodes statiques `readAllBytes` (pour un fichier binaire) et `readAllLines` pour les fichiers texte.

Ainsi, le code précédent peut s'écrire de manière plus compacte :

```
Charset UTF8 = Charset.forName("UTF-8");
Path codeMain =  Paths.get("./src/scolarite/Main.java");

try {
        List<String> lignes = Files.readAllLines(codeMain,UTF8);
        for (String ligne: lignes)
                System.out.println(ligne);
}
catch (IOException ioe) {
        System.err.println("Erreur de lecture : "+ioe);
}
```

Lecture d'un flux L'interface `InputStream` modélise les flux en lecture, i.e. les données que l'on peut lire mais qui arrivent au fil et à mesure. Ceci est le cas typiquement de communications réseau ou lorsque l'on attend des données saisies au clavier (`System.in`).

On peut analyser le flux de données arrivant d'un objet `InputStream` à l'aide de la classe `Scanner`.

Exemple de lecture de données au clavier :

```
InputStream is = System.in; // le flux is est un flux en lecture sur le clavier
Scanner sc = new Scanner(is); // sc est un analyseur attache au flux is
System.out.println("Entrez un nombre au clavier");
int reponse = sc.nextInt(); // on recupere le premier entier arrivant du flux is
System.out.println("Votre nombre est "+reponse);
```

Exemple de lecture d'un fichier texte contenant des nombres :

```java
Path fichier =  Paths.get("./unFichier.txt");
InputStream is = null;
try {
 is = Files.newInputStream(fichier);
 Scanner sc = new Scanner(is);
 while (sc.hasNextInt())
    System.out.println(sc.nextInt());
  if (is != null)
    is.close();
}
catch (IOException ioe) {
 System.err.println("Echec lecture du fichier : "+ioe);
}
```

Exercice 68. *(⋆) Tester ce code sur un fichier dont le contenu est :*

```
10 378 38397
3673
3367 happy
end
```

Modification du contenu d'un fichier La modification du contenu d'un fichier est analogue à l'opération de lecture. On utilise un objet implémentant l'interface `BufferedWriter`.

Exemple de code pour créer un fichier texte encodé en `UTF8` :

```java
Charset UTF8 = Charset.forName("UTF-8");
Path fichier =  Paths.get("./unFichier.txt");
try {
        BufferedWriter bw = Files.newBufferedWriter(fichier,UTF8);
        bw.write("Ceci est \n un essai.");
        bw.write("\nImpressionnant, non?");
        bw.flush(); // ecrire toutes les donnees en attente (vider le tampon)
        bw.close();
}
catch (IOException ioe) {System.err.println("Erreur d'ecriture : "+ioe);}
```

Après exécution, le fichier `test.txt` est créé avec le contenu :

```
Ceci est
 un essai.
Impressionnant, non?
```

Exercice 69. *(⋆) Tester.*

Exercice 70. *(⋆) Dans la classe `Personne`, écrire une méthode `void save(String)` qui sauvegarde la personne dans le fichier dont le nom est passé en paramètre.*

Exercice 71. *(⋆) Ecrire une méthode statique `Personne load(String)` qui réalise l'opération inverse : elle retourne une personne initialisée à partir du contenu du fichier dont le nom est passé en paramètre.*

Ecriture dans un flux L'interface `OutputStream` modélise les flux en écriture, i.e. les données que l'on envoie par le biais d'un flux. Ceci concerne les communications réseau mais aussi l'affichage dans un terminal (`System.out`).

La classe `PrintWriter` est adaptée à l'écriture de données de type texte, et peut être rattachée à un flux en écriture.

Exemple d'écriture dans un fichier texte en utilisant un flux :

```java
Charset UTF8 = Charset.forName("UTF-8");
Path fichier =  Paths.get("/home/gothmog/test.txt");
OutputStream os;
try {
        os = Files.newOutputStream(fichier);
        PrintWriter pw = new PrintWriter(os); // pw est rattache au flux os
        pw.write("Ceci est \n un essai.");
        pw.write("\nImpressionnant, non?");
        pw.flush(); // ecrire toutes les donnees en attente (vider le tampon)
        if (os != null)
      os.close();
    pw.close();
}
catch (IOException ioe) {System.err.println("Echec ecriture: "+ioe);}
```

Fichiers de paramètres : la classe `Properties` Les fichiers sont souvent utilisés pour sauvegarder un ensemble de paramètres. Par exemple, pour une personne avec un age de 23 ans, de nom "Cloclo" et prénom "Claudette", on peut enregistrer son état sous la forme :

```
nom=Cloclo
prenom=Claudette
age=23
```

La classe `Properties` modélise les ensembles de paramètres. Ses principales méthodes sont :
- `String getProperty(String key)` : retourne la valeur associée à `key` ;
- `void load(InputStream is)` : charge un ensemble de paramètres à partir d'un flux ;
- `void store(OutputStream out, String comments)` : enregistre l'ensemble des paramètres dans un flux ;
- `Object setProperty(String key, String value)` : ajoute le paramètre `key=value` ;

Exercice 72. *(⋆⋆) Créez l'ensemble de paramètres correspondant à l'exemple ci-dessus, puis enregistrez le dans un fichier texte. Consultez avec un éditeur de texte le contenu de ce fichier. Réciproquement, récupérez en Java les valeurs des paramètres stockés dans ce fichier.*

Parcours d'un répertoire La méthode statique `newDirectoryStream` de la classe `Files` permet de parcourir l'ensemble des noms de fichiers et des sous-répertoires contenu dans un répertoire.

Par exemple, ce code affiche l'ensemble du contenu du répertoire `/home/gothmog` :

```java
Path dir =  Paths.get("/home/gothmog/");
try {
    DirectoryStream<Path> fichiers = Files.newDirectoryStream(dir);
    for (Path p:fichiers)
        System.out.println(p);
    if (fichiers != null)
            fichiers.close();
}
catch (IOException ioe) {System.err.println("Echec lecture: "+ioe);}
```

Exercice 73. *(⋆) Tester sur votre répertoire personnel.*

5.2.2 Réseau

En Java, effectuer des communications réseaux se fait de manière analogue à lire ou écrire dans des fichiers.

Dans cette partie, nous abordons brièvement le cas de la programmation réseau dans le cadre du réseau Internet (IP), avec les protocoles de transport TCP ou UDP.

Exceptions spécifiques En raison des particularités des communications réseaux, les incidents déclenchent des exceptions spécifiques (héritant de `IOException`), dont :
- `SocketException` : erreur lors de la création d'une socket ;
- `MalformedURLException` : URL de syntaxe incorrecte ;
- `SSLException` : erreur dans le protocole SSL (cryptage) ;
- `RemoteException` : erreur du côté de la machine distante.

Adressage IP : la classe `InetAddress`

Définition 5.1 (InetAddress)**.** *Classe permettant de gérer les adresses IP et les noms associés.*

On peut créer une instance de cette classe à partir d'une machine dont on connait le nom.

Exemple :

```java
InetAddress ia = InetAddress.getByName("info-ssh2.iut.u-bordeaux1.fr");
```

Outre la méthode statique `getByName`, cette classe offre plusieurs autres méthodes statiques utiles :
- `public static InetAddress getLocalHost()` : adresse IP de la machine locale ;
- `public static InetAddress[] getAllByName(String hostname)` : toutes les adresses IP d'une machine dont on connait le nom :

A partir d'une instance de `InetAddress`, on peut
- connaitre le nom de la machine : `public String getHostName()`
- récupérer les 4 octets de l'adresse : `public byte[] getAddress()`

Exemple :

```java
System.out.println("adresse IP du serveur web du LaBRI : "+InetAddress.getByName("www.labri.fr
    "));
```

Exercice 74. *(⋆) Affichez le nom et les adresses IP de votre machine.*

Protocole de transport UDP Le protocole UDP (User Datagram Protocol) est un protocole rapide de transport des données, sans connexion, mais non-fiable (des paquets peuvent être perdus et/ou ne pas arriver dans l'ordre dans lequel ils ont été transmis).

Définition 5.2 (DatagramPacket). *Cette classe permet de créer des objets qui contiendront les données d'un datagramme UDP.*

Deux constructeurs sont disponibles, l'un pour les paquets à recevoir, l'autre pour les paquets à envoyer :
— `public DatagramPacket(byte buffer[], int taille)` : cela construit un objet pour recevoir un datagramme ; le paramètre *taille* correspond à la taille maximale des datagrammes à recevoir.
— `public DatagramPacket(byte buffer[], int taille, InetAddress a, int p)` : ce constructeur crée un objet pour envoyer un datagramme, à la machine d'adresse *a* sur le port *p*.

Définition 5.3 (DatagramSocket). *Cette classe permet de créer des "sockets" (prises) UDP pour l'envoi et la réception des datagrammes UDP.*

Ses principales méthodes sont :
— `public DatagramSocket(int port)` : constructeur créant un objet attaché au port UDP local passé en paramètre ;
— `public void send(DatagramPacket data)` : pour envoyer le datagramme *data* ;
— `public void receive(DatagramPacket data)` : pour réceptionner un datagramme, stocké dans *data*.

Exemple d'envoi d'un message

```
InetAddress address = InetAddress.getByName("raoul.labri.fr");
int port = 4321;
String ch = "Le message"; int chl = ch.length();
byte[] message = new byte[chl]; ch.getBytes(0, chl, message, 0);

DatagramPacket dp = new DatagramPacket(message, chl, address, port);
DatagramSocket ds = new DatagramSocket();
ds.send(dp);
```

Exemple de réception de messages

```
byte[] buffer = new byte[1024]; String ch;
DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
DatagramSocket ds = new DatagramSocket(4321);
ds.receive(dp);
ch = new String(buffer, 0, 0, dp.getLength());
System.out.println("Paquet recu : message = " + ch +
        " - envoyeur = " + dp.getAddress().getHostName() +
        " - port = " + dp.getPort());
```

Exercice 75. *(⋆⋆) Ecrire 2 classes `ClientUDP.java` et `ServeurUDP.java` permettant le transfert d'un message de l'un à l'autre, caractère par caractère. Mesurer le temps de transmission. La transmission de 1000 messages est-elle fiable ?*

TCP Le principe général pour une communication TCP/IP est d'établir une connexion, puis ensuite de rattacher à celle-ci un flux en lecture (resp. écriture) de type `InputStream` (resp. `OutputStream`) pour recevoir (resp. émettre) des données.

Définition 5.4 (la classe `Socket`). *Elle permet de gérer et exploiter une connexion TCP.*

On crée une instance de cette classe en passant en paramètre le nom de la machine distante et le port de la machine distante sur lequel la socket (prise) doit être rattachée :

```
Socket s = new Socket(String nomMachineDistante, int portDistant);
```

A partir d'une instance, on peut récuperer des flux en lecture ou écriture, pour recevoir ou transmettre des données, via les deux méthodes :

```
public InputStream  getInputStream()  throws IOException;
public OutputStream getOutputStream() throws IOException;
```

En outre, la classe `Socket` expose un certain nombre d'informations sur la connexion, via les méthodes :

```
public InetAddress getInetAddress(); // adresse IP machine distante
public InetAddress getLocalAddress(); // adresse IP machine locale
public int getPort(); // port de la machine distante
public int getLocalPort(); // port de la machine locale
```

La méthode `close()` ferme la connexion et libère les ressources du système associées à la connexion.

Exemple de client TCP

```
try {
    Socket s = new Socket("nom machine distante",1234);
    OutputStream os = s.getOutputStream();
    InputStream  is = s.getInputStream();
    os.write((int)'a'); // envoi du caractere a sous la forme d'un entier
    System.out.println(is.read());
    s.close();
} catch(Exception e) {
    // Traitement d'erreur
}
```

La machine distante doit être en écoute lorsque la machine locale demande à établir une connexion. La classe `ServerSocket` permet de prendre en charge les demandes de connexion.

Elle prend en paramètre le port sur lequel la machine écoute et le nombre maximal de demande de connexions qui peuvent être placées sur la file d'attente. Lorsqu'une demande de connexion est acceptée, celle-ci peut être prise en charge par une instance de la classe `Socket` en utilisant la méthode `accept` de la classe `ServerSocket`.

Exemple de serveur TCP

```
try {
    ServerSocket ecoute = new ServerSocket(1234,5); // ecoute sur le port 1234, taille de la
       file d'attente = 5
    Socket service = (Socket)null;
    while(true) {
        service = ecoute.accept(); // une demande de connexion a ete acceptee - l'objet
           service de la classe Socket la prend en charge
        OutputStream os = service.getOutputStream();
        InputStream  is = service.getInputStream();
        os.write(is.read()); // on renvoie ce qui a ete lu (serveur: "ping pong")
        service.close();
    }
} catch (Exception e) {
    // traitement d'erreur
}
```

Exercice 76. *(⋆) Ecrire 2 classes `ClientTCP.java` et `ServeurTCP.java` permettant le transfert d'un message de l'un à l'autre, caractère par caractère. Mesurer le temps de transmission. La transmission de 1000 messages est-elle fiable ?*

5.2.3 Sérialisation

La sérialisation est un procédé permettant de mettre un objet sous une forme à partir de laquelle il pourra être reconstitué à l'identique. Ceci permet de sauvegarder des objets sur disque, ou bien de les transmettre via le réseau. En sérialisant un objet, on peut donc créer un objet persistant, i.e. dont la durée de vie n'est pas liée à celle de la JVM. Le format utilisé est indépendant du système d'exploitation.

La sérialisation utilise les classes `ObjectOutputStream` et `ObjectInputStream` et l'interface `Serializable`.

L'interface `Serializable` Cette interface ne définit aucune méthode. Tout objet qui doit être sérialisé doit implémenter cette interface. Si l'on tente de sérialiser un objet qui n'implémente pas l'interface `Serializable`, une exception `NotSerializableException` est levée.

La classe `ObjectOutputStream` Cette classe permet d'obtenir un flux en écriture pour écrire/envoyer des objets sérialisés.

Exemple de sauvegarde d'un objet sérialisé `o` dans un fichier `objet.ser`

```
Path p = Paths.get("objet.ser");
try {
        OutputStream os = Files.newOutputStream(p);
        ObjectOutputStream oos = new ObjectOutputStream(os);
        oos.writeObject(o);
        oos.flush();
        oos.close();
}
catch (IOException e) {System.err.println(e);}
```

Exercice 77. *(⋆) Ecrire une méthode `save2(String nomFichier)` dans la classe `Personne` qui enregistre dans un fichier l'objet sous forme sérialisée.*

Toutes les variables de la classe à sérialiser, doivent-elles même être sérialisables. Par exemple, si l'objet contient une collection d'éléments, chaque élément de la collection doit être sérialisable.

Exercice 78. *(⋆) Ecrire une méthode `save2(String nomFichier)` dans la classe `Groupe` qui enregistre dans un fichier tout un groupe sous forme sérialisée.*

La classe `ObjectInputStream` Cette classe permet d'obtenir un flux en entrée pour lire/recevoir des objets sérialisés.

Exemple de lecture d'un objet de type `Personne` stocké dans le fichier `objet.ser`

```
Path p = Paths.get("objet.ser");
Personne res=null;
try {
        InputStream is = Files.newInputStream(p);
        ObjectInputStream ois = new ObjectInputStream(is);
        res=(Personne) ois.readObject();
        ois.close();
}
catch (IOException e) {System.err.println(e);}
catch (ClassNotFoundException e) {System.err.println(e);}
```

Exercice 79. *(⋆) Tester le code ci-dessus en écrivant une méthode statique `load2` dans la classe `Personne` retournant une personne.*

Exercice 80. *(⋆) Ecrire une méthode `public static Groupe load2(String nomFichier)` dans la classe `Groupe` qui charge tout un groupe stocké sous forme sérialisé.*

Notez que dans le cas d'objets complexes comme ceux de la classe `Groupe`, la sérialisation est un mécanisme particulièrement puissant et simple d'emploi.

Fiche de synthèse

Flux en Java
- flux en entrée : interface `InputStream`
- flux en sortie : interface `OutputStream`
- traitement des erreurs d'entrée/sortie : `try ... catch(IOException){...}`
- flux d'octets : `FileInputStream`, `FileOutputStream`
- flux de caractères : `FileReader`, `FileWriter`
- flux avec tampon : `BufferedReader`, `BufferedWriter`
- manipulation de chemin : la classe `Path`
- fichiers de paramètres : la classe `Properties`

Programmation réseau
- adressage IP : la classe `InerAddress`
- transport UDP : les données `DatagramSocket`, la transmission `DatagramSocket`
- transport TCP : la connexion côté client `Socket` , la mise en place de flux via les méthodes `getInputStream()` et `getOutputStream()`, l'écoute côté serveur `ServerSocket`

Sérialisation
- l'interfacce `Serializable`
- les flux via les classes `ObjectInputStream` et `ObjectOutputStream`

Objets distants via RMI
- côté serveur : l'interface `Remote`
- côté client : la demande d'accès via la classe `Registry` et sa méthode `lookup`

Java et XML
- les évènements : `XMLEvent`
- le parcours des évènements, des attributs

Exemple de lecture d'un fichier texte contenant des nombres :

```
Path fichier =  Paths.get("./unFichier.txt");
try {
 InputStream is = Files.newInputStream(fichier);
 Scanner sc = new Scanner(is);
 while (sc.hasNextInt())
 System.out.println(sc.nextInt());
} catch (IOException ioe) {
 System.err.println("Echec lecture du fichier : "+ioe);
 }
```

Exemple d'écriture dans un fichier texte en utilisant un flux :

```
Charset UTF8 = Charset.forName("UTF-8");
Path fichier =  Paths.get("./unFichier.txt");
OutputStream os;
try {
        os = Files.newOutputStream(fichier);
        PrintWriter pw = new PrintWriter(os); // pw est rattache au flux os
        pw.write("Ceci est \n un essai.");
        pw.write("\nImpressionnant, non?");
        pw.flush(); // ecrire toutes les donnees en attente (vider le tampon)
} catch (IOException ioe) {System.err.println("Echec ecriture: "+ioe);}
```