Chapitre VI : Les flux en Java
Interractions avec le système d'exploitation

Eric.Leclercq@u-bourgogne.fr

Département IEM
http://ufrsciencestech.u-bourgogne.fr
http://ludique.u-bourgogne.fr/~leclercq

3 février 2017

Plan

Principes et définitions

- Les entrées / sorties sont organisées en Java autour du concept de flux ou flot (*stream*)

Définition :

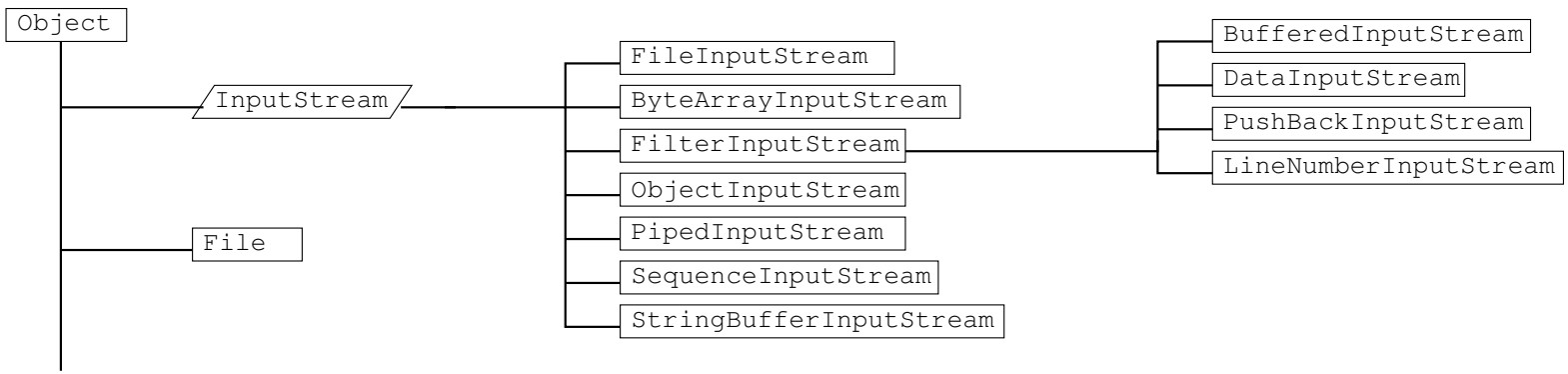
Un flux est un canal de communication entre un lecteur et un rédacteur.

- Les flux sont classifiés en :
  - flux d'entrée qui comportent des méthodes de lecture ;
  - flux de sortie qui comportent des méthodes d'écriture.
- Un flux d'entrée est connecté à une source, un flux de sortie à une cible.
- La source ou la cible d'un flux peuvent être un fichier, un tampon en mémoire, une chaîne de caractères, une connexion réseau, un autre flux.

Organisation de l'API

- L'API Java propose une infrastructure de flux basée sur les packages `java.io` et `java.nio`.
- Les flux sont des objets
- Tous les flux sont des descendants des classes (abstraites) `InputStream`, `OutputStream`, `Reader` et `Writer`.
- Les flux élémentaires sont des flux d'octets (flux binaires).
- Les classes pour les flux non structurés (séquences d'octets) en écriture, respectivement lecture, sont `OutputStream` respectivement `InputStream` (suffixes).
- Les classes pour les flux de type séquences de caractères *Unicode*, en écriture, respectivement lecture, sont `Writer` et respectivement `Reader` (suffixes).
- Les classes `InputStreamReader` et `OutputStreamWriter` sont des ponts entre les flux structurés et non structurés.

Les classes construites à partir d'`InputStream`



Lecture et écriture des flux binaires

InputStream et OutputStream sont les classes de base qui définissent l'interface de plus bas niveau des flux d'octets.

Les méthodes générales de lecture sur un flux d'octets sont :

- `int read()` : lit l'octet suivant disponible sur le flux (se bloque en l'attendant) et retourne la valeur lue dans un `int` (valeur entre 0 et 255), retourne -1 si la fin du flux est atteinte ;
- `int read(byte b[])` lit au plus `b.length` octets depuis le flux, les placent dans le tableau `b`, et retourne le nombre d'octets lus ou bien -1 si la fin du flux est atteinte.

Les méthodes générales d'écriture sur un flux d'octets sont :

- `void write(int c)` écrit un octet, représenté par un int
- `void write(byte[] b)` écrit les `b.length` octets de b

Entrées-sorties de type terminal

La classe `java.lang.System` défini les E/S standard : `in`, `out`, `err`.

- Par comparasion avec C on peut établir la correspondance :

```
1 InputStream stdin = System.in;
2 OutputStream stdout = System.out;
3 OutputStream stderr = System.err;
```

- `out` et `err` ne sont pas réellement des `OutputStream` mais des `PrintStream` spécialisés et très utiles pour l'affichage.
- Il est possible de lire un caractère sur l'entrée standard avec la méthode `read()` et de tester la fin du flux (valeur retournée `-1`) :

```
1 try{ int val = System.in.read(); }
2 catch (IOException e) {}
3 ...
4 byte b=(byte) val;
```

Entrées-sorties de type terminal

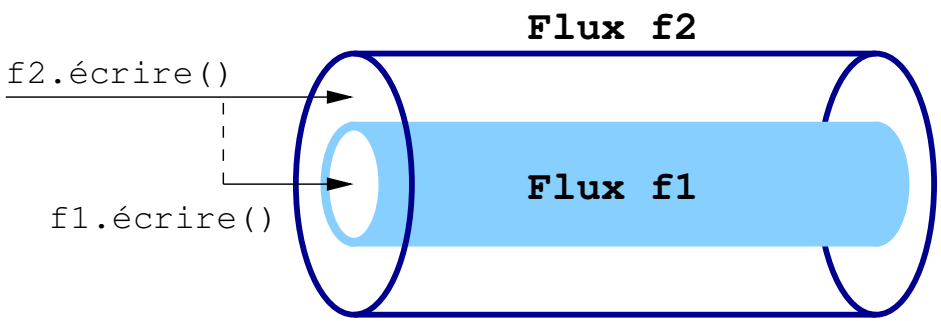
- Toutes les opération de lecture et d'écriture sur les flux peuvent déclencher une exception `IOException`.
- la méhtode `available()` permet de vérifier le nombre d'octets disponibles lors de la lecture pour créer ensuite la structure avec la taille adaptée.

```
1  int attente= System.in.available();
2  byte data[]=new byte[attente];
3  System.in.read(data);
```

- la méthode `close()` libère les ressources système allouées.
- `InputStream` propose la méthode `skip()` pour sauter un certain nombre de d'octets.

Les enveloppes de flux

- Ce principe consiste à ajouter des fonctionnalités au flux.
- Le plus souvent il s'agit de transformations ou de filtrage.
- Un flux filtre prendre le flux cible comme argument de son constructeur et lui délègue les appels avec avoir effectué ses opération de filtrage

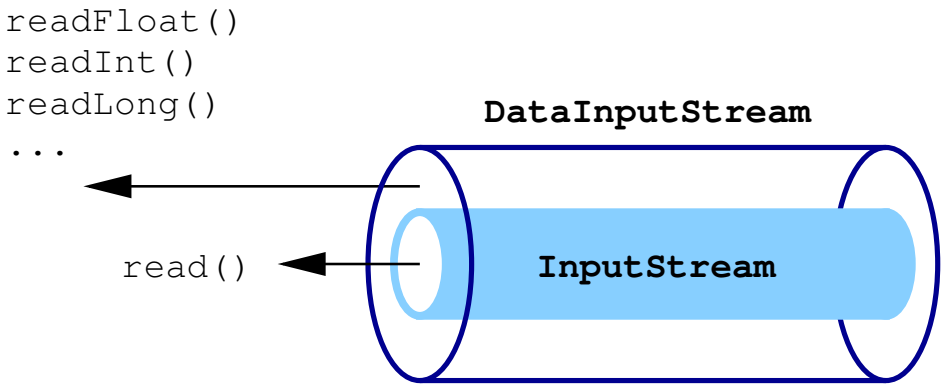


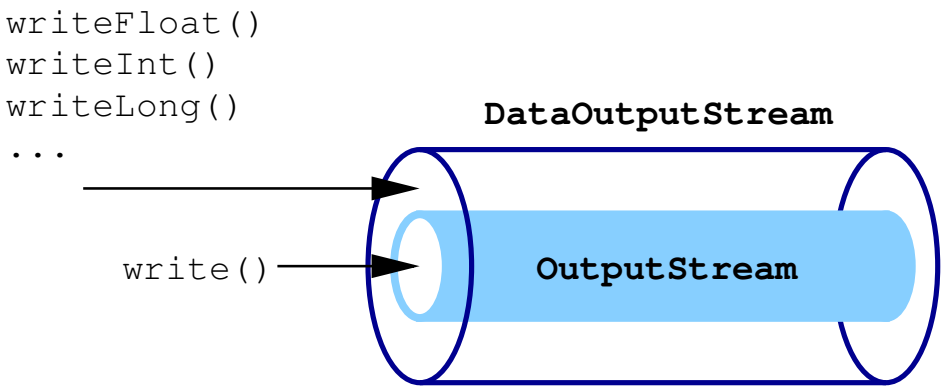
Pont entre les flux de caractères et d'octets

- `InputStreamReader` et `OutputStreamWriter` sont des flux de caractères enveloppant un flux d'octets.
- Un schéma d'encodage permet une conversion dans les deux sens.
- Le modèle de d'encodage peut être donné en paramètre au constructeur.

```
1  try{
2   InputStreamReader convert = new InputStreamReader(System.in);
3   // on enveloppe dans BufferedReader pour bénéficier de readLine()
4   BufferedReder in = new BufferedReader(convert);
5   String teste=in.readLine();
6   int i = NumberFormat.getInstance().parse(texte).intValue();
7   }
8   catch(IOException e){...}
9   catch(ParseException pe){...}
```

Pont entre les flux de caractères et d'octets



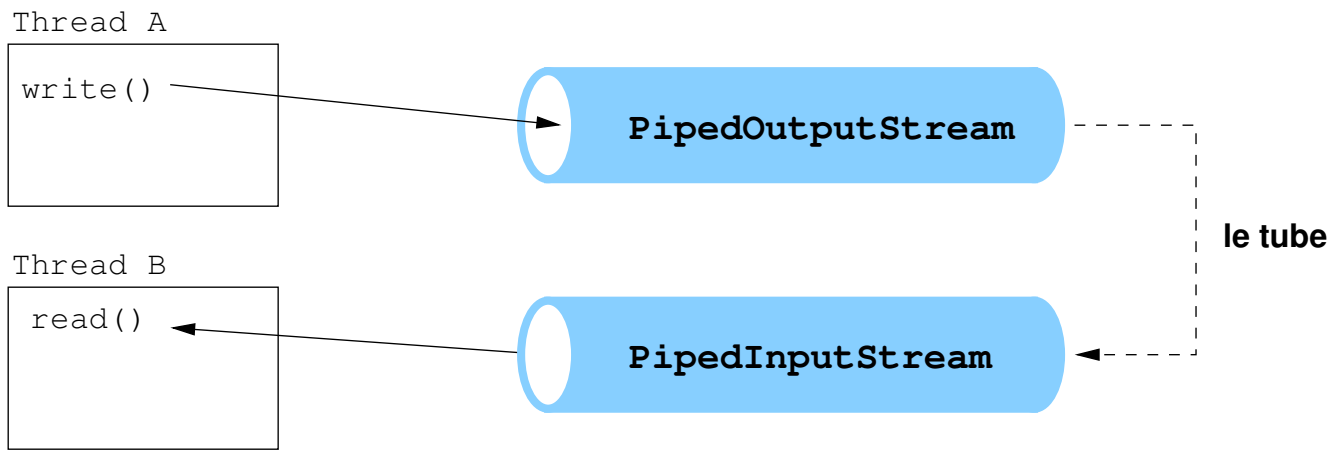


Les tubes (*pipes*)

Définition :

`PipedOutputStream` et `PipedInputStream` permettent de connecter les extémités des flux

Les pipes permettent de faire communiquer les thread sans passer par des structures globales (**static**)



Les tubes (*pipes*)

- Exemple de création et connexion de tubes :

```
1 PipedInputStream tubeEntree = new PipedInputStream();
2 PipedOutputStream tubeSortie = new PipedOutputStream(tubeEntree);
```

- L'ordre inverse de création est aussi possible ;
- Il est possible de créer les tubes séparément et de les connecter ensuite dynamiquement au moyen de la méthode `connect()` ;
- Les flux `PipedReader` et `PipedWriter` sont réservés aux flux de caractères ;
- Si le *buffer* interne du tube est plein le processus qui écrit est bloqué et mis en attente jusqu'a ce que le processus qui lit est fait de la place ;
- De façon symétrique, le processus lecteur est bloqué si le *buffer* est vide.

Les tubes (*pipes*) : un exemple

Processus de lecture d'un flux (fichier) :

```
1  import java.io.BufferedReader;
2  import java.io.IOException;
3  import java.io.PrintWriter;
4  public class Lecteur extends Thread {
5    private PrintWriter out = null;
6    private BufferedReader in = null;
7    public Lecteur(PrintWriter out, BufferedReader in){
8     this.out = out;
9     this.in = in;
10   }
11   public void run() {
12    if (out != null && in != null){
13    try {
14     String input;
15     while ((input = in.readLine()) != null){
16       out.println(input.toUpperCase());
17       out.flush();
18     }
19     out.close();
20    } catch (IOException e) {e.printStackTrace(); }
21   }
22  }
23 }
```

Les tubes (*pipes*) : un exemple

Processus connecté au tube :

```
1  import java.io.BufferedReader;
2  import java.io.FileReader;
3  import java.io.PipedReader;
4  import java.io.PipedWriter;
5  import java.io.PrintWriter;
6  public class ExempleTube {
7   public static void main(String[] args) throws Exception {
8    FileReader f=new FileReader("toto.txt");
9    BufferedReader in=new BufferedReader(f);
10
11   PipedWriter pipeOut = new PipedWriter();
12   PipedReader pipeIn = new PipedReader(pipeOut);
13
14   PrintWriter out = new PrintWriter(pipeOut);
15   new Lecteur(out, in).start();
16
17   BufferedReader bin = new BufferedReader(pipeIn);
18   String input;
19
20   while ((input = bin.readLine()) != null){
21    System.out.println(input);}
22   in.close();
23  }
24 }
```

Connecter un flux à un fichier (exemple)

- La lecture et l'écriture d'octets dans un fichier se fait à l'aide des classes `FileInputStream` et `FileOutputStream`

```
1  import java.io.*;
2  class CopieBinaireFichier {
3    public static void main(String[] args)
4      throws IOException {
5      ...
6      int c;
7      ...
8      if (args.length > 0) in = new FileInputStream(args[0]);
9      if (args.length > 1) out = new FileOutputStream(args[1]);
10     while ((c = in.read()) != -1) out.write(c);
11     in.close();
12     out.close();
13   }
```

Association de flux pour la gestion des fichiers

Définition :

`DataInputStream` et `DataOutputStream` sont des flux filtrants qui permettent de lire ou d'écrire des chaînes de catactères et des types de base codés sur plusieurs octets.

- `DataInputStream` et `DataOutputStream` implémentent les interfaces `DataInput` et `DataOutput`.
- Elles offrent des méthodes plus évoluées que `FileOutputStream` et `FileInputStream`
- Elles proposent des méthodes pour envoyer des types primitifs dans le flux : `writeInt` etc.
- L'exemple suivant montre l'enrichissement, l'association et la connexion de flux.
- Un objet de type `file` peut être utilisé dans les constructeurs de flux.

Association de flux pour la gestion des fichiers

Exemple d'association avec le flux d'entrée standard

```
1 DataInputStream dis = new DataInputStream(System.in);
2 double d = dis.readDouble();
```

Exemple d'association avec un fichier

```
1 ...
2 FileOutputStream f = new FileOutputStream("fic.dat");
3 DataOutputStream sortie = new DataOutputStream(f);
4 ...
5 DataOutputStream sortie = new DataOutputStream(
6         new FileOutputStream("fic.dat");
7 ...
```

Obtenir les propriétés d'un fichier

- `java.io.File` est utile pour obtenir diverses propriétés de fichiers :
  - savoir si un chemin désigne un fichier ordinaire ou un répertoire
  - savoir si l'objet est accessible en lecture ou en écriture, etc.

```
1  File cheminRepertoire =
2    new File("/usr/local/java/docs");
3  File cheminFichier =
4    new File(cheminRepertoire, "index.html");
5  ...
6    if (cheminRepertoire.isDirectory() &&
7        cheminFichier.canRead()) {
8      ...
9    }
```

- Les constructeurs de fichier ne lèvent pas d'exception.

Obtenir les propriétés système

Il est possible de consulter les propriétés système (*System Properties*) : `System.getProperty("clé");`. La clé peut prendre les valeurs suivantes :

- `file.separator`
- `line.separator`
- `user.dir`
- `java.version`
- `java.home`
- `java.class.path`
- etc.

Opérations principales sur les fichiers

Méthode	Type retour	Description
`canRead()`	`Boolean`	le fichier ou le répertoire est-il accessible en lecture ?
`canWrite()`	`Boolean`	le fichier ou le répertoire est-il accessible en écriture ?
`createNewFile()`	`Boolean`	Crée un nouveau fichier
`delete()`	`Boolean`	détruit le fichier ou le répertoire
`exists()`	`Boolean`	est-ce que le fichier ou le répertoire existe ?
`getAbsolutePath()`	`String`	renvoie le chemin absolu du fichier ou du répertoire
`getName()`	`String`	renvoie le nom du fichie ou du répertoire
`getParent()`	`String`	renvoie le nom du répertoire parent du fichier ou du répertoire
`getPath()`	`String`	renvoie le chemin du fichier ou du répertoire
`isAbsolute()`	`Boolean`	le chemin du fichier ou du répertoire est-il absolu
`isDirectory()`	`Boolean`	s'agit-il d'un répertoire ?
`isFile()`	`Boolean`	s'agit-il d'un fichier ?
`lastModified()`	`long`	renvoie la date de la dernière modification du fichier ou du répertoire
`length()`	`long`	renvoie la taille du fichier
`list()`	`String []`	renvoie la liste des fichiers du répertoire
`listfiles()`	`File []`	renvoie la liste des fichiers du répertoire sous la forme d'un tableau d'objet de type `File`
`mkdir()`	`Boolean`	crée un répertoire
`mkdirs()`	`Boolean`	crée tous les répertoires du chemin
`renameTo(File dest)`	`Boolean`	renome le fichier ou le répertoire
`setLastModified()`	`Boolean`	définit l'heure de la dernière modification du fichier ou du répertoire
`setReadOnly()`	`Boolean`	définit le fichier en lecture seule
`toURL()`	`java.net.URL`	génère un objet URL pour ce fichier ou répoertoire

Fichiers à accès direct

- La classe `RandomAccessFile` peut être utilisée si un accès direct à une position quelconque du fichier est nécessaire
- Fonctionne à la fois en écriture et en lecture
- Un cuseur permet de se déplacer dans le fichier
- Il faut lui adjoindre un index pour trouver les éléments
- Il faut être capable de déterminer (calculer) la position d'un éléments en fonction de sa taille
- Le constructeur `RandomAccessFile(String name, String mode)` s'emploie avec une chaîne de caractères contenant le nom de fichier suivi du mode d'accès `r` pour lecture ou `rw` pour lecture et écriture.
- Le constructeur `RandomAccessFile(File file, String mode)` fait de même à partir d'un objet `File`

Fichiers à accès direct

- Pour naviguer dans le fichier :
  - utiliser un pointeur qui indique la position dans le fichier ou les opérations de lecture ou de mise à jour doivent être effectuées ;
  - la méthode `getFilePointer()` permet de connaître la position du pointeur
  - la méthode `seek()` permet de déplacer le pointeur : en paramètre un entier long qui réprésentent la position, dans le fichier, précisée en octets (la première position commence à zéro).

```
1  import java.io.RandomAccessFile;
2  public class TestRandomAccesFile {
3    public static void main(String[] args) {
4      try {
5        RandomAccessFile monFichier = new RandomAccessFile("monfichier.dat", "rw")
             ;
6        monFichier.seek(5*4); // x * sizeof int
7        System.out.println(monFichier.readInt());
8        monFichier.close();
9      } catch (Exception e) {
10       e.printStackTrace();
11     }
12   }
13 }
```

Lire et écrire sur un flux de caractères

- Les flux de caractères sont des objets de classe `Reader` ou `Writer`
- Les méthodes `read` et `write` de ces classes sont analogues à celles opérant sur des flux d'octets, à la différence que c'est un caractère 16 bits qui est lu ou écrit, et non un octet
- La conversion entre un flux d'octets et un flux de caractères se fait au moyen des classes `OutputStreamWriter` et `InputStreamReader`
- Le principe est l'association de flux : connecter un flux d'octets à un flux de caractères
  - `InputStreamReader isr = new InputStreamReader(System.in);`
- Pour connecter un flux de caractères à un fichier, si la conversion n'est pas nécessaire, il est plus simple de recourir aux classe `FileWriter` et `FileReader`

Connecter un flux à un tampon

- Les opérations individuelles de lecture ou d'écriture peut être très coûteuses
- C'est le cas des accès à un fichier, ou des accès au réseau
- Pour éviter des opérations individuelles sur un octet ou sur un caractère à la fois, on utilise un tampon (*buffer*)
- Pour écrire sur un fichier on écrira dans un flux bufferisé
- Les classes qui mettent en oeuvre les buffers sont :
  - `BufferedInputStream` et `BufferedOutputStream` pour les flux binaires
  - `BufferedReader` et `BufferedWriter` pour les flux de caractères

Connecter un flux à un tampon

- Un flux de caractères de la classe `BufferedReader` propose des opérations supplémentaires (par exemple, lecture d'une ligne de texte).

```
BufferedReader in =
  new BufferedReader(new FileReader("toto"));
String s = in.readline();
```

- De façon similaire pour écrire dans un fichier, il est préférable de travailler avec un tampon :

```
PrintWriter out =
  new PrintWriter(
    new BufferedWriter(
      new FileWriter("toto")));
out.println("un long texte");
```

Lire et écrire des données brutes sur un flux binaire

- pour lire et écrire, non pas des octets, mais des valeurs d'un type connu, il faut utiliser les classes `DataInputStream` et `DataOutputStream`
- Ces classes disposent de méthodes spécialisées pour les types primitifs : `readInt()`, `readDouble()`, `readChar()`, `readBoolean()` (et les méthodes writeXYZ correspondantes)

```
DataOutputStream dos =  new DataOutputStream(
    new FileOutputStream('toto.dat'));
...
dos.writeBoolean(false);
dos.writeInt(400);
...
dos.close();
```

- Même exemple pour lire un entier sur l'entrée standard :

```
DataInputStream dis =
  new DataInputStream(System.in);
int n = dis.readInt();
```

Lire et écrire des données par blocs

- Les flux peuvent être utilisés afin de lire ou d'écrire des données par bloc (tableau d'octets)
- Supposons que l'on dispose d'un tableau d'octets `data` reçu par exemple depuis une connexion réseau :
  - on sait que ce tableau contient un booléen et un entier
  - on utilise les flux des classes `DataInputStream` et `ByteArrayInputStream`

```
byte[] data = ...;
DataInputStream dis = new DataInputStream(
    new ByteArrayInputStream(data));
...
boolean b = dis.readBoolean();
int n = dis.readInt();
dis.close();
```

Lire et écrire des données par blocs

- Le tableau doit continir des données codées de façon compatible avec le décodage réalisé
- Ce sera le cas si le tableau d'octets a été obtenu par les classes `DataOutputStream` et `ByteArrayOutputStream`

```
byte[] data;
ByteArrayOutputStream baos = new ByteArrayOutputStream();
DataOutputStream dos = new DataOutputStream(baos);
dos.writeBoolean(true);
dos.writeInt(4);
data = baos.toByteArray();
dos.close();
```

Principes

- Les classes `ObjectOutputStream` et `ObjectInputStream` permettent de rendre persistants des objets de Java en les sauvegardant au travers d'un flux associé à un fichier
- Si un objet comporte des membres qui sont des références à d'autres objets ces objets persistent aussi.
- Afin de pouvoir sérialiser un objet d'une classe donnée, celle-ci doit implémenter l'interface `Serializable` ou hériter d'une classe elle-même sérialisable.
- L'interface `Serializable` ne possède aucun attribut et aucune méthode, elle sert uniquement à identifier une classe sérialisable.
- Tous les attributs de l'objet sont sérialisés mais à certaines conditions :
  - être lui-même sérialisable ou être un type primitif
  - ne pas être déclaré `static`
  - ne pas être déclaré à l'aide du mot clé transient

Exemple de création

- Un `serialVersionUID` est un numéro de version, associé à toute classe implémentant l'interface Serializable.
- il permet de s'assurer que lors de la désérialisation les versions des classes Java sont concordantes.
- Si le test échoue, une `InvalidClassException` est levée.
- Une classe sérialisable peut déclarer explicitement son `serialVersionUID`
- Si une classe sérialisable ne déclare pas explicitement un `serialVersionUID`, Java en calcule un par défaut.
- Il est fortement recommandé de déclarer explicitement le `serialVersionUID`.

```
private static final long serialVersionUID = 123L;
ObjectOutputStream s =
  new ObjectOutputStream(new FileOutputStream("toto.dat"));
s.writeObject("Aujourd'hui");
s.writeObject(new Date());
```

Exemple de restitution

- La lecture de ces objets se fait dans le même ordre que leur écriture
- doit opérer une coercition (transtypage ou casting) du type `Object` vers la classe que l'on veut restituer

```
ObjectInputStream s =
  new ObjectInputStream(new FileInputStream("toto"));
String chaine = (String)s.readObject();
Date date = (Date)s.readObject();
```

Un exemple

- Unflux peut aussi être associé à une connexion réseau et permettre d'envoyer des objets sérialisables au travers tu réseau (chapitre 8)
- Le programme exemple suivant connecte un flux d'entrée à une URL (vue comme un fichier)
- Il transforme ce flux d'octets en un flux de caractères et le place dans un tampon
- Les lignes successivement lues en entrée sont copiées sur la sortie standard (jusqu'à ce que `readLine` retourne null)

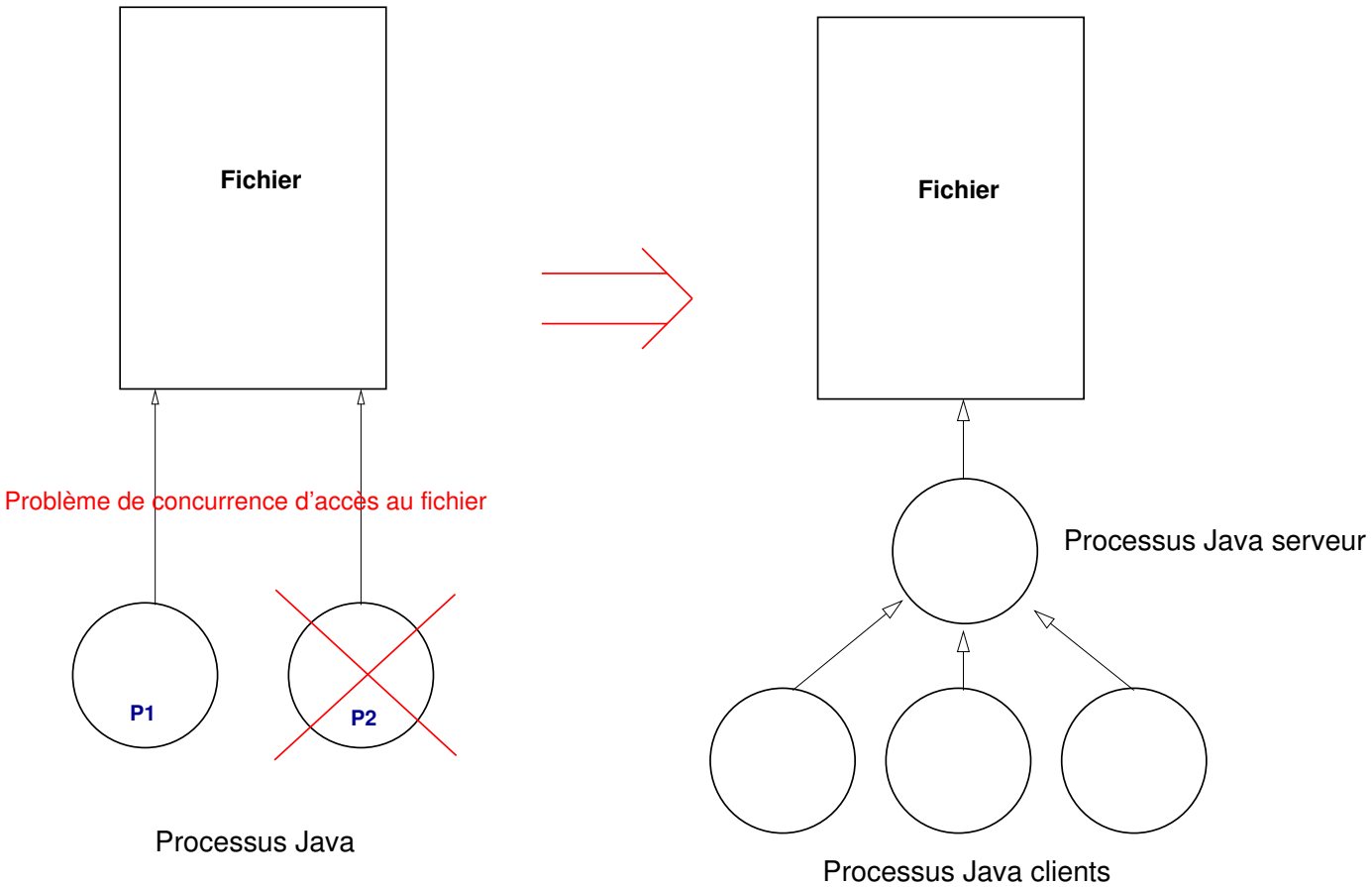
Un exemple

```
1  import java.net.*;
2  import java.io.*;
3  class URLReader {
4    public static void main(String[] args)
5      throws MalformedURLException, IOException {
6      URL url = new URL("http://kundera/index.html");
7      BufferedReader in =
8        new BufferedReader(
9          new InputStreamReader(
10     url.openStream()));
11     String ligne;
12     while ((ligne = in.readLine()) != null)
13       System.out.println(ligne);
14     in.close();
15   }
16 }
```

Pour continuer...

- Comment se comporte le mécanisme de sérialisation vis à vis de l'héritage ?
- Comment spécialiser ou modifier la sérialisation ?
- La classe `Hashtable` permet de stocker des valeurs en les associant à des identificateurs uniques calculés à partir des clés :
  - Les clés et valeurs ne doivent pas possèder la valeur null, mais en revanche peuvent être des objets de n'importe quel type
  - $\Rightarrow$ sérialiser le hastable pour gérer un fichier de type base de données pouvant contenir n'importe quel type d'objet Java.
  - Comparaison `Hashtable` et `Hashmap`, solutions ?
- Comment se préserver des accès concurrents sur les fichiers ?

Accès concurrents sur les fichiers



Java nio

- le package `java.nio` est proposé depuis la version 1.4 de Java ;
- nio = new IO (nouvelles E/S) ;
- Propose des E/S non bloquantes et sélectionnables
- À l'origine un thread s'occupe d'une E/S
- Avec nio un thread peut prendre en charge plusieurs canaux d'E/S et tester la disponibilité de données (comme en C)
- Permet l'abandon et l'interruption des E/S donc de reveiller ou stopper un thread bloqué sur une E/S

Le modèle d'E/S de nio

- Le modèles d'E/S du package nio est construit autour du concept de *buffer* ;
- Les buffer de nio sont des tableaux améliorés destinés à la communication ;
- nio propose la notion de *buffer* directs : la mémoire associée réside hors de la machine virtuelle Java càd directement dans le système d'exploitation ;
- nio propose deux caractéristiques génériques liées aux fichiers : le mappage en mémoire et le verouillage ;
- Les fichier mappés se maniuplent comme si ils étaient résident en permanence en mémoire ;
- Le verouillage s'articule autout des verrous partagés ou exclusifs sur des zones de fichiers.

Le modèle d'E/S de nio

- Le package nio propose la notion de canal (subsitituée à celle de flux) ;

Définition :

Un canal est un terminal de comminucation semblable aux flux mais plus abstrait dans le sens ou l'interface élémentaire des canaux ne précise rien quant aux communications mais définit les méthodes `isOpen()` et `close()`.

- Les implémentation des canaux dédiées aux fichiers, sockets ou à des périphériques quelconques ajoutent leurs propres méthodes ;
- nio propose les canaux princpaux suivant : `FileChannel`, `Pipe.SinkChannel`, `Pipe.SourceChanel`, `SocketChannel`, `ServerSocketChanel` etc.

Canaux fichiers

- Le canal `FileChannel` est l'équivalent de `RandomAccessFile` ;
- Un `FileChannel` est construit à partir d'un `FileInputStream`, d'un `FileOutputStream` ou d'un `RandomAccessFile` ;

```
1 FileChannel readOnlyfs = new FileInputStream("fic.txt").getChannel();
2 FileChannel readWritefc = new RandomAccessFile("fic.txt", "rw").getChannel
      ();
```

- Un canal a un statut ouvert à sa création ;

Canaux fichiers

- L'utilisation d'un `FileChannel` se fait à travers un buffer de type `ByteBuffer` ;

```
1  try {
2       ReadableByteChannel channel = new FileInputStream("infile").
            getChannel();
3       // Create a direct ByteBuffer
4       ByteBuffer buf = ByteBuffer.allocateDirect(10);
5       int numRead = 0;
6          while (numRead >= 0) {
7              // read() places read bytes at the buffer's position so the
8              // position should always be properly set before calling read
                   ()
9              // This method sets the position to 0
10             buf.rewind();
11             // Read bytes from the channel
12             numRead = channel.read(buf);
13             // The read() method also moves the position so in order to
14             // read the new bytes, the buffer's position must be set back
                    to 0
15             buf.rewind();
16             // Read bytes from ByteBuffer
17             for (int i=0; i<numRead; i++) {
18                 byte b = buf.get();
19             }
20         }
21     } catch (Exception e) {
22     }
```

Verouillage

- Les `FileChannel` supportent les verrous exclusifs ou partagé sur des zones de fichiers
- Les verrous sont gérés par la méthode `lock()`

```
1  FileLock verrouFichier = canalf.lock();
2  int debut = 0;
3  int longueur = canalf2.size();
4  FileLock verrouLecture = canalf2.lock(début,longueur,true);
```

- Les verrous peuvent être patagés ou exclusifs :
  - un verrou exclusif empêche quiconque d'obtenir un verrou sur le fichier ou sur une zone ;
  - un verrou partagé permet à d'autres d'acquérir des verrous partagés mais non des verrous exclusif : lorqu'on écrit, on interdit toutes les écritures. Lorsqu'on lit, on interdit les écritures concurrentes, les lectures concurrentes restent possibles.

Verouillage

- La méthode `lock()` permet d'obetnir un verrou exclusif ;
- La seconde forme (avec paramètres) permet d'indiquer une longueur et un drapeau indiquant si le verrou est partagé ou exclusif ;
- L'objet retourné par la méthode (`fileLock`) sert ensuite à relacher le verrou au moyen de la méthode `release()`.
- Très utile pour des interactions lecteurs rédacteurs.

Verouillage

```
1  public class lock{
2  public static void main(String arg[]){
3     try {
4         // Get a file channel for the file
5         File file = new File("lock.java");
6         FileChannel channel = new RandomAccessFile(file, "rw").getChannel();
7
8         // Use the file channel to create a lock on the file.
9         // This method blocks until it can retrieve the lock.
10        FileLock lock = channel.lock();
11
12        // Try acquiring the lock without blocking. This method returns
13        // null or throws an exception if the file is already locked.
14        try {
15            lock = channel.tryLock();
16        } catch (OverlappingFileLockException e) {
17            // File is already locked in this thread or virtual machine
18        }
19
20       Reader reader = new InputStreamReader(System.in);
21       BufferedReader input = new BufferedReader(reader);
22       System.out.print("Appuyez␣sur␣Entree␣");
23       String prenom = input.readLine();
24       // Release the lock
25       lock.release();
26
27        // Close the file
28        channel.close();
29    } catch (Exception e) {
30    }
```