Les flots Java (Streams)

Les entrées/sorties Java sont bâties sur des flux (flot/streams). Le programme Java lit les données sur un InputStream et il écrit les données sur un OutputStream. Il existe plusieurs types des flux représentés par des classes de flux différentes:

- Les classes de flux **ByteArrayInputStream** et **ByteArrayOutputStream** permettent respectivement de lire et d'écrire des données dans un tableau de bytes,
- Les classes de flux **Java.io.FileInputStream** et **Java.io.FileoutputStream** permettent respectivement de lire et d'écrire dans un fichier,
- Les classes de flux **Java.io.InputStream** et **Java.io.OutputStream** permettent respectivement de lire et d'écrire sur les flux qui représentent la connexion TCP

Ce sont les mêmes méthodes de bases qui sont utilisables pour lire les données sur tous les types d'InputStreams ou écrire les données sur tous les types d'OutputStreams.

L'écriture sur un OutputStream

Les principales méthodes nécessaires à l'écriture de données sur un OutputStream sont les suivantes.

1. *Public abstract void write(int b) throws IOException*
2. *Public abstract void write(byte[] data) throws IOException*
3. *Public abstract void write(byte[] data, int offset, int length) throws IOException*
4. *Public abstract void flush() throws IOException*
5. *Public abstract void close() throws IOException*

La méthode fondamentale d'OutputStream est write(int b). Cette méthode utilise un entier (4 octets) pour coder une valeur comprise entre 0 et 255. Pour la valeur 255 (00000000 0000000

00000000 11111111), seul l'octet de poids faible est écrit sur le flux de sortie. Les 3 autres octets sont ignorés.
Les méthodes 2 et 3 sont multi octets. Elles permettent d'écrire un tableau d'octets sur le flux de sortie. La méthode 3 permet d'écrire sur le flux, l*ength* octets du tableau data à partir de la position *offset*.

La lecture sur un InputStream

Les principales méthodes nécessaires à la lecture de données sur un InputStream sont les suivantes :

1. *Public abstract void int read( ) throws IOException*
2. *Public abstract void int read(byte[] input) throws IOException*
3. *Public abstract void int read(byte[] data, int offset, int length) throws IOException*
4. *Public abstract void close() throws IOException*

La méthode de base d'un flux d'entrée InputStream est la méthode 1 read(). Cette méthode lit 1 seul octet de données sur le flux d'entrée source InputStream) et le retourne dans un entier (int) dont la value vaut entre 0 et 255.

- Dans le cas de lecture d'un flux de type réseau, la méthode est bloquante en attente d'un caractère à lire sur le flux.
- Dans le cas d'une lecture d'un flux de type FileInputStream (fichier) lorsque la fin de fichier est rencontrée (EOF), la méthode read() retourne la valeur -1.

Pour être plus performant, on peut lire plusieurs octets à la fois et les placer dans un tableau d'octets. Les méthodes 2 ou 3 sont multi-octets. Elles retournent le nombre d'octets effectivement lus sous forme d'un entier. La méthode 2 cherche à remplir le tableau. La méthode 3 lit *length* octets et les place dans le tableau data[] à partir de la position offset. Si aucun octet n'est disponible, alors que le flux n'est pas fermé, la valeur retournée sera égale à 0.

La gestion des fichiers

Les fichiers sont gérés à l'aide de flux. Un fichier que l'on ouvre en écriture dans un programme est un flux FileOutputStream. Un fichier que l'on ouvre en lecture sera vu par le programme comme un flux d'entrée de type FileInputStream.

L'écriture dans un fichier (File OutputStream)

L'exemple montre comment on peut écrire dans le fichier fic1, les 10 octets contenus dans le tableau d'octets data1 à partir de la position n°2. Tout d'abord, Il faut spécifier le chemin d'accès vers le répertoire dans lequel se trouve le fichier fic1. Les sous-classes de FileOutputStream utilisent les méthodes des OutputStream, décrites précédemment, pour écrire des données sur le disque.

Exemple d'écriture d'un fichier
La classe utilisée est java.io.FileoutputStream.

public abstract classe FileOutputStream

```
byte[] data1= new byte[20];
String s= "CECI EST UN ESSAI";
data1= s.getBytes("ASCII");
FileOutputStream fea= new FileOutputStream ("d:\\temp\\fic1");
fea.write(data1,2,10);
fea.close();
      }
catch(IOException ex) {System.out.println(ex);}
```

La lecture dans un fichier (FileInputStream).

Dans cet exemple, read() lit un octet dans un entier b et puis le stocke dans un tableau d'octets input[]. Lessous-classes de FileInputStream utilisent des méthodes des InputStream, vues précédemment, pour lire des données sur le fichier.

Exemple de lecture de fichier

La classe d'entrée de base de Java est java.io.FileInputStream.

public abstract classe FileInputStream

```
byte[] input= new byte[10];
             int b;
             int i;
             try{
FileInputStream fe= new FileInputStream("d:\\temp\\test.txt");
for ( i=0;i<input.length;i++){
              b= fe.read();
                    if (b==-1) break;
             input[i]=(byte)b;
             }
System.out.println(" tableauinput : "+input[0]+" " +input[1]+" " +input[2]+" "
+input[3]);
fe.close();
      }
catch(IOException ex) {System.out.println(ex);}
Remarque :
```

Attention avant de stocker b dans un byte à l'aide d'un transtypage, il faudrait s'assurer que sa valeur n'excède pas +127 (1 byte permet de stocker un entier signé de valeur comprise entre -127 et +127).

Les filtres de flot (filter streams)

Les flux InputStream et OutputStream sont des classes brutes (raw). Elles lisent ou écrivent des octets seuls ou en groupes mais c'est à peu près tout. Les filter streams permettent de créer des buffers, de lire ou écrire du binaire pour une autre interprétation que des bytes java. Nous en détaillons deux qui vous seront utiles en TP.

Les filtres DataStream

Les classes DataOutputStream et DataInputStream fournissent des méthodes pour lire et écrire des types primitifs Java dans un format binaire. Toutes les données sont écrites ou lues en complément à 2 avec le minimum d'octets possibles :

- 1 long sur 8 octets.
- 1 int sur 4 octets.
- 1 short sur 2 octets.
- 1 byte sur 1 octet.
- 1 Char sur 2 octets non signés.
- 1 Booléen sur 1 octet contenant la valeur 0 (false) ou 1 (true).

DataInputStream est complémentaire de DataOutputStream: **tout ce qui a pu être écrit sera lu.**

Le flux de données de sortie (DataOutputStream)

DataOutputStream offre 11 méthodes pour écrire des types java particuliers : entier, caractères, octets, booléen… Nous en citerons 7.

1. Public final void writeInt(int c) throws IOException
2. Public final void writeShort(int c) throws IOExceptio

3. Public final void writeByte(int c) throws IOException
4. Public final void writeChar(int c) throws IOException
5. Public final void writeBoolean(boolean b) throws IOException
6. Public final void writeBytes(String s) throws IOException
7. Public final void writeChars(String s) throws IOException

La méthode WriteBytes itère à partir l'argument String mais écrit seulement la partie la moins significative de chaque caractère c'est-à-dire 1 octet.
La méthode WriteChars itère avec l'argument String mais écrit chaque caractère de la chaine sous forme de 2 octets big-endian unicode character.

Flux de données en entrée(DataInputStream)

Il existe 9 méthodes pour lire des données en binaire. Nous citerons :

1. Public final int readInt() throws IOException
2. Public final short readShort() throws IOException
3. Public final byte readByte() throws IOException
4. Public final boolean readBoolean() throws IOException.
5. **Public final int readUnsignedByte() throws IOException**
6. **Public final int readUnsignedShort() throws IOException**
7. Public final int read(Byte[] input) throws IOException
8. Public final int read(Byte[] input, int offset, int length) throws IOException
9. Public final int readFully(Byte[] input) throws IOException
10. Public final int readFully(Byte[] input, int offset, int length) throws IOException

Ces méthodes lisent en binaire sur le flux de données et retournent des entiers, des entiers short, des octets, des booléens…
Les méthodes 5 et 6 permettent de **lire des bytes et des short non signés** et retournent les entiers (integer) équivalents.
Les méthodes multi-Bytes 7 et 8 lisent plusieurs octets sur un flux, puis les placent dans un tableau et retournent le nombre d'octets lus.
Les méthodes readFully permettent de lire les données jusqu'à ce que le nombre de bytes demandé (Length) soit atteint et de les placer dans le tableau à partir de la position offset. Si ce n'est pas le cas, une exception est lancée. Cette méthode permet de lire un nombre attendus d'octets. *Par exemple lorsque le HTTP header content_length donne le nombre exact d'octets reçus dans le fichier*.

Exemple
Si *input1* est un InputStream, on peut déclarer un filtre in1, qui permettra de lire directement des short (sur 2 octets) au lieu de bytes.

```
short[] in_short= new short[2];

DataInputStream in1 = new DataInputStream (input1);
      in_short[0]= in1.readShort();
      in_short[1]=in1.readShort();
```

La méthode readLine()

La classe DataInputStream propose la très populaire méthode **readLine()** qui lit une ligne terminée par un caractère de terminaison de ligne et retourne une chaîne de caractères.

Public final void String readLine() throws IOException

Toutefois cette méthode ne devrait pas être utilisée dans toutes circonstances car elle a un petit problème: elle reconnait 2 caractères comme fin de ligne CR et LF. En effet, readLine() reconnait soit

- un seul caractère LF (= 10, 0aH, " \r")
- la paire de caractères CR (=13, 0dh, "\n ") **suivi de LF** (= 10, 0aH, " \r").

Si la ligne se termine par LF tout va bien, le caractère LF est supprimé et la ligne est retournée au programme sous forme d'un String.
Lorsque le caractère CR est détecté, readLine() attend le caractère suivant avant de continuer:

- si c'est LF alors CR et LF sont supprimés et la ligne est retournée comme un String.
- Si le caractère qui arrive n'est pas un LF, la ligne est transmise comme un String **et le caractère reçu fait partie de la nouvelle ligne.**

Ainsi si le caractère CR (0dh) est le tout dernier caractère de la ligne, readLine() attend un caractère qui n'arrivera pas pour retourner la chaine de caractères lue. On peut citer en exemple, certains vieux protocoles réseau, le flux réseau provenant de Macintosh ou le flux fichier crée sur un Macintosh qui se termine par un seul caractère ODH.

La méthode ReadLine() doit être utilisée sur un flux de type InputStreamReader (qui peut être chaîné sur un flux TCP déjà bufferisé.

Le filtre BufferedStream

Il permet au programme de créer une zone de stockage (buffer) dans lequel il viendra lire ou écrire.

BufferedInputStream

2 constructeurs:

public BufferedInputStream (InputStream in)
public BufferedInputStream (InputStream in, int buffersize)

Le premier argument est le flux d'entrée et le second la taille du buffer. Par défaut celle-ci est de 2048 octets.
Lorsque le programme lit des données, il les lit dans le buffer jusqu'à ce le buffer soit vide. Ensuite il lira autant de données qu'il peut sur le flux pour remplir le buffer, même s'il ne les utilise pas tout de suite. Lorsqu'on lit un fichier, on a intérêt à «bufferiser» le flux pour augmenter les performances de lecture.

BufferedOutputStream

2 constructeurs

public BufferedOutputStream (OutputStream out)
public BufferedOutputStream (OutputStream out, int buffersize)

Le premier argument est le nom du flux de sortie et le second la taille du buffer. Par défaut celle-ci est de 512 octets. Lorsqu'un OutputBuffer est déclaré, les octets écrits par le programme sont stockés dans le buffer jusqu'à ce que celui-ci soit plein ou bien que le programme exécute un flush sur le flux.

Les conversions de données

Elles sont souvent nécessaires lorsqu'on reçoit des données provenant du réseau qui sont interprétées directement par le langage java. Par exemple, on reçoit un octet de valeur supérieure à 127 et java l'interprète comme un nombre négatif (car un byte est signé en Java).

Lorsqu'on dispose d'un flux d'entrée (InputStream) ou de sortie (OutputStream), on peut faire les conversions en utilisant les DataInputStream ou DataOutputStream (voir § ci-dessus). En dernier lieu,

si on ne dispose pas d'un flux, les conversions peuvent toujours se faire en connaissant le principe de codage utilisé par Java.

- 1 entier est écrit sur 4 octets signés.
- 1 short est écrit sur 2 octets signés.
- 1 octet est écrit sur 1 octet signés.
- 1 Char est écrit sur 2 octets non signés.

Conversion d'un octet vers un entier

Lorsque l'on déclare un byte en java, on réserve un octet dans lequel on peut coder un entier signé dont la valeur varie de -128 à +127. Le signe est codé dans l'octet de poids fort du byte. Si on veut déclarer une constante dont la valeur est supérieure à +127, Java nous donne une erreur de compilation.

Mais lorsqu'on reçoit (du réseau) un octet qui est logé un byte du programme et qui a une valeur supérieure à +127, celui-ci est interprété comme une valeur négative. Le problème pour nous c'est de lui redonner sa « vraie» valeur positive. Pour cela on va utiliser un **transtypage**.

Un transtypage force le type indiquant au compilateur que l'on est conscient du problème de risque d'information. Certains transtypages sont implicitement autorisés, sachant qu'au cours d'opérations utilisant des zones différentes: le compilateur réservera toujours la plus grande zone.

byte→short→int→long→float→double

Supposons que l'on reçoive du réseau une valeur +128 codée 1000 0000 : on peut la transférer dans une zone int de 4 octets où elle sera codée -128 en prenant le complément à 2 (vrai) de ce nombre.

Le complément à 2 s'obtient en ajoutant 1 au complément restreint.
Le complément restreint s'obtient en en inversant les bits constituant le nombre.

- Soit la valeur absolue 00000000 00000000 00000000 1000000 +128
- le complément restreint 11111111 11111111 11111111 0111111
- Rajouter 1 1
- Résultat -128 11111111 11111111 11111111 1000000

Pour « récupérer » la valeur positive +128, on fait un « et logique » avec 0x000000FF ;
11111111 11111111 11111111 10000000 -128
00000000 00000000 00000000 11111111 0x000000FF
00000000 00000000 00000000 10000000 +128

byte Zone1 ;
Int Zone2 ;
Zone1= (byte) Zone2 ;

```
Byte LSB=data_rec[1];
LSB=LSB&0x000000FF;
Byte MSB=data_rec[0];
MSB= MSB&0x000000FF;
int no_bloc= MSB*256+LSB;
```

Pour «repasser» en mode byte avant d'envoyer l'octet, on fait un transtypage de int vers byte : pour forcer Java à transférer l'octet de poids faible de l'entier vers l'octet que nous voulons obtenir.

Conversion de 2 octets en integer

Pour convertir 2 octets qui codent une valeur positive comprise entre 1 et 65535. Il faut considérer chacun des octets séparément pour le transcoder dans un integer comme ci-dessus.

Octet poids fort (O1)	Octet poids faible (O2)

Une fois obtenue la valeur absolue de O1 dans int1 et celle de O2 dans int2 on peut reconstituer la valeur reçue dans O3←O1*256+O2.

Conversion d'un short en byte.

Il suffit de considérer que sur un short, l'octet de poids le plus faible peut avoir une valeur comprise entre 0 et 255. Si la valeur contenue dans le short est inférieure ou égale à 255, il suffit de faire un transtypage de short vers byte. Sinon, on peut faire le transtypage du premier octet puis isoler le 2° octet (et logique), diviser la valeur obtenue par 256 et faire le transtypage vers le 2° octet. L'exemple montre comment l'entier signé de valeur comprise entre 0 et +32767 est placé dans les octets data[0] et data[1] .

Exemple

```
byte[] data= new  byte [2];
int num= 200;
data[0]=0;
data[1]=(byte)num;
if (num>255){ num= (short)(num&0x0000FF00);
              data[0]= (byte)(num/256);}
```

Le résultat :

`data[0] 0 et data[1] -56`

La valeur d'un byte est comprise entre -127 et+127. Cette valeur peut être interprétée comme valeur positive (par exemple dans le cas d'un échange de numéro de blocs avec TFTP).

+56= 32+16+8 =00111000
Cplt rst = 11000111
+1 1
-56 = 11001000

Ce qui donne data[1]= +200 en valeur absolue.