ITI 1521. Introduction informatique II

Laboratoire 9

Hiver 2007

[PDF]

Objectifs

- Introduction aux entrées-sorties (E/S) en Java
- Approfondir les notions liées aux exceptions

Introduction

Ce laboratoire comporte deux parties. La première partie introduit les concepts de base des entrées-sorties qui seront nécessaires pour réaliser ce laboratoire (conservez ces notes et ces exemples, ils vous seront utiles pour le cours structures de données). La seconde partie consiste à modifier l'application **PlayListManager** afin de lire/écrire les chansons à partir de fichiers.

1 Entrées-sorties (E/S) en Java

Ce document présente les éléments de base pour faire des entrées-sorties (E/S) en Java. Il couvre un sous-ensemble des classes du package **java.io**. Depuis la version 1.4 de Java, il y a un nouveau package, **java.nio** (new io), définissant des concepts plus avancés d'E/S tels que les «buffers», «channels», et «memory mapping», ces sujets ne seront pas couverts ici.

Les entrées-sorties en Java semblent assez complexes à première vue. Tout d'abord, il y a un très grand nombre de classes. Ensuite, il faut combiner plusieurs objets pour réaliser les traitements. Pourquoi cette complexité ? Java est un langage moderne, développé au milieu des années 1990s alors que le Web allait devenir une réalité. Ainsi, les données peuvent être lues et écrites à partir de plusieurs sources, dont le clavier, la console, des disques externes, mais aussi le réseau. De plus, la présence du Web a aussi stimulée la création de classes permettant l'internationalisation de programmes (pour les postes de travail anglophones, pour les langues européennes, mais aussi arabes et orientales) ainsi que le traitement de données multimédia.

1.1 Définitions

Un **flux** (stream) est une séquence ordonnée de données ayant une source ou une destination. Il y a deux genres de flux : les **flux de caractères** (character streams) et les **flux d'octets** (byte streams).

Java utilise des Unicodes pour encoder les caractères — les flux de caractères sont en général associés aux entrées-sorties de textes (donc lisible par l'humain). Les flux de caractères s'appellent **readers** et **writers**. Ce document traite principalement de ces types de flux. Les flux d'octets (byte streams) sont associés aux entrées-sorties de données (binaires). Les fichiers audio et vidéo, jpeg et mp3, en sont des exemples. Les informations peuvent être lues ou écrites sur un support externe. Pour chaque flux en lecture (ou reader), il existe un flux en écriture (ou writer) correspondant. Il existe

aussi un troisième mode d'accès, le mode **direct**, permettant à la fois les lectures et écritures. Le mode direct n'est pas traité ici.

1.2 Exposé général

Il existe deux genres de flux et trois modes d'accès.
- **Flux :** caractères ou octets ;
- **Accès :** lecture, écriture ou direct.

De plus, le support utilisé (clavier, console, disque, mémoire, réseau, etc.) impose aussi ses propres contraintes (faut-il un tampon (buffer) ou pas, par exemple). Le package **java.io** comporte quelque 50 classes, 10 interfaces et plus de 15 exceptions. Le grand nombre de classes pourrait à lui seul intimider les nouveaux programmeurs. Pour ajouter à cette complexité, il faut généralement combiner des objets de deux ou trois classes afin d'effectuer quelque traitement que ce soit, comme le démontre cet exemple.

```
InputStreamReader in = new InputStreamReader( new FileInputStream( "data" ) );
```

ici "data" est le nom d'un fichier d'entrée. Les sections qui suivent passent en revue les principaux concepts liés aux entrées-sorties en Java. La majorité des concepts sont accompagnés d'exemples et d'exercices. Compilez et exécutez tous les exemples. Complétez tous les exercices.

1.3 Flux

InputStream et **OutputStream** sont deux classes abstraites définissant les méthodes communes aux flux d'entrée et de sortie.

1.3.1 InputStream

La classe **InputStream** déclare les trois méthodes suivantes.
- **int read() :** Lit le prochain octet du flux d'entrée. L'octet lu est retourné dans un entier, intervalle 0 à 255. Si aucun octet n'est disponible, signifiant que la fin du flux a été atteinte, alors la méthode retourne la valeur -1.
- **int read(byte[] b) :** Lit plusieurs octets à la fois. Les octets sont sauvegardés dans le tampon (buffer, un tableau) **b**. La méthode retourne le nombre d'octets lus.
- **close() :** Fermeture du flux d'entrée. Libère les ressources qui lui sont associées.

La classe **InputStream** est abstraite. Voici des exemples de ses sous-classes : **AudioInputStream**, **ByteArrayInputStream**, **FileInputStream**, **FilterInputStream**, **ObjectInputStream**, **PipedInputStream**, **SequenceInputStream** et **StringBufferInputStream**. Parmi celles-ci, la classe **FileInputStream** sera présentée ci-bas. Cette classe permet la lecture d'octets à partir d'un fichier.

1.3.2 OutputStream

La classe abstraite **OutputStream** déclare les méthodes qui suivent.
- **write(byte[] b) :** Écrit **b.length** octets sur la sortie.
- **flush() :** Vide la mémoire tampon du flux, forçant ainsi l'écriture de tous les octets se trouvant encore dans le tampon.
- **close() :** Fermeture du flux. Libère les ressources associées à ce flux.

La classe **OutputStream** est abstraite. Voici des exemples de sous-classes concrètes : **ByteArrayOutputStream**, **FileOutputStream**, **FilterOutputStream**, **ObjectOutputStream** et **PipedOutputStream**. Parmi celles-ci, la classe **FileOutputStream** est utilisée fréquemment et sera étudiée ci-bas. Elle permet l'écriture de l'octet dans un fichier.

1.3.3 System.in et System.out

Deux objets sont prédéfinis par le système. **System.in** est un flux d'entrée, généralement associé au clavier. **System.out** est un flux de sortie, généralement associé à la console.

1.4 Lecture

Limitons la portée de cette discussion à la lecture à partir d'un fichier ou la lecture à partir du clavier.

1.4.1 Lecture à partir d'un fichier

Afin de lire des données d'un fichier, il faut créer un objet **FileInputStream**. Attardons-nous aux deux constructeurs suivants.

- **FileInputStream(String name) :** Ce constructeur reçoit le nom du fichier en paramètre. Exemple.
  ```
  InputStream in = new FileInputStream( "data" );
  ```
- **FileInputStream(File file) :** Ce constructeur reçoit en paramètre un objet **File**, un objet représentant le fichier externe.

```
File f = new File( "data" );
InputStream in = new FileInputStream( f );
```

L'objet **File** permet d'effectuer toutes sortes d'opérations sur le fichier. Voici quelques exemples.

```
f.delete();
f.exists();
f.getName();
f.getPath();
f.length();
```

FileInputStream est une sous-classe d'**InputStream**. Tout comme son parent, cette classe ne lit que des octets.

1.4.2 InputStreamReader

La classe **FileInputStream** sert de passerelle entre un flux d'octets et un flux de caractères. On l'utilise comme suit.

```
InputStreamReader in = new InputStreamReader( new FileInputStream( "data" ) );
```

ou encore,

```
InputStreamReader in = new InputStreamReader( System.in );
```

L'objet **System.in** est généralement associé au clavier du poste de travail.

- **int read() :** Lecture d'un caractère. Retourne -1 lorsque la fin de l'entrée est atteinte (end-of-file (eof), end-of-stream (eos)). L'entier doit être converti en caractère.
  ```
  int i = in.read();
  if ( i != -1 ) {
      char c = (char) i;
  }
  ```
  Voir Unicode.java et Keyboard.java.
- **int read(char [] b) :** Lit plusieurs caractères à la fois. Les caractères sont mis dans le tableau **b**. La méthode retourne le nombre de caractères lus ou **-1** si la fin est atteinte.
  ```
  InputStreamReader in = new InputStreamReader( new FileInputStream( "data" ) );

  int i = in.read();
  if ( i != -1) {
      char c = (char) i;
  }
  ```

ou

```
char[] buffer = new char[ 256 ];
num = in.read( buffer );
String str = new String( buffer );
```

Exercice 1 *Concevez une classe afin de lire des caractères au clavier utilisant la méthode* **read(char[] b)***; le nombre de caractères lus est déterminé par la taille du tableau (tampon). Utilisez la classe* ***Keybord*** *comme point de départ.*

Faites quelques tests. Peu importe le nombre de caractères lus, la longueur de la chaîne est toujours 256. De plus, certains symboles ne sont pas affichables. Vous devez utiliser la méthode **trim** afin de retirer les caractères non affichables.
Voir Keyboard.java.

1.4.3 BufferedReader

Certaines applications doivent lire les données ligne par ligne. Pour ces applications, nous utiliserons un (objet) **BufferedReader**. BufferedReader utilise un objet de la classe InputStreamReader afin de lire les données. Ce dernier, InputStreamReader utilise InputStream afin de lire les octets. Chaque couche (objet) ajoute de nouvelles fonctions. InputStreamReader convertit les octets en caractères. Finalement, BufferedReader regroupe les caractères en chaînes de caractères, par exemple une ligne à la fois.

```
FileInputStream f = FileInputStream( "data" );
InputStreamReader is = new InputStreamReader( f );
BufferedReader in = new BufferedReader( is );
```

ou

```
BufferedReader in = new BufferedReader(
                        new InputStreamReader(
                            new FileInputStream("data") ) );
String s = in.readLine();
```

Voir Copy.java.

Exercice 2 *Créez un programme affichant toutes les lignes d'un fichier contenant un certain mot. Affichez aussi le numéro de la ligne.*

Solution : Find.java

Exercice 3 *Écrivez un programme afin de compter le nombre d'occurrences d'un mot donné dans un fichier.*

Exercice 4 *Implémentez une classe afin de télécharger et afficher le contenu d'une page Web.*

Solution : WGet.java

1.5 Écriture

Considérons maintenant l'écriture sur sortie standard ainsi que l'écriture dans un fichier. Vous remarquerez la similarité avec la lecture.

1.5.1 Écrire dans un fichier

Afin d'écrire dans un fichier, nous utiliserons un (objet) **FileOutputStream**. Voici deux constructeurs.

- **FileOutputStream(String name) :** Crée un flux de sortie pour l'écriture dans un fichier nommé **name**.
  ```
  OutputStream out = new FileOutputStream( "data" );
  ```
- **FileOutputStream(File file) :** Ce constructeur reçoit un objet **File**.
  ```
  File f = new File( "data" );
  OutputStream out = new FileOuputStream( f );
  ```
  Tout comme son parent, **OutputStream**, cette classe ne sert qu'à l'écriture d'octets.
- **OutputStreamWriter :** Une passerelle pour la conversion de caractères en octets.
  ```
  OutputStreamWriter out = new OutputStreamWriter( new FileOutputStream( "data " ) );
  ```
  ou
  ```
  OutputStreamWriter out = new OutputStreamWriter( System.out );
  OutputStreamWriter err = new OutputStreamWriter( System.err );
  ```
  Les messages d'erreurs sont en général écrits sur **System.err**, la sortie standard. Voici les méthodes de la classe **OutputStreamWriter**.
  - **write(int c) :** Écrit un seul caractère ;
  - **write(char[] buffer) :** Écrit le contenu du tableau sur la sortie ;
  - **write(String s) :** Écriture d'une chaîne de caractères.

  Exercice 5 *Modifiez le programme Copy.java afin de spécifier un fichier destination. Ainsi, l'application copie le contenu d'un fichier d'entrée dans un fichier sortie.*
  Solution : Copy.java
- **PrintWriter :** Cette classe définit un ensemble de méthodes permettant l'écriture de valeurs d'un type primitif ou objet.
  ```
  print( boolean b )  : Prints a boolean value.
  print( char c )     : Prints a character.
  print( char[] s )   : Prints an array of characters.
  print( double d )   : Prints a double-precision floating-point number.
  print( float f )    : Prints a floating-point number.
  print( int i )      : Prints an integer.
  print( long l )     : Prints a long integer.
  print( Object obj ) : Prints an object.
  print( String s )   : Prints a string.
  ```
  Le méthodes suivantes affichent aussi un séparateur de lignes (le séparateur varie selon le système d'exploitation utilisé, cette difficulté est traitée pour nous par l'objet PrintWriter).
  ```
  println()             : Prints a line separator string.
  println( boolean b )  : Prints a boolean value.
  println( char c )     : Prints a character.
  println( char[] s )   : Prints an array of characters.
  println( double d )   : Prints a double-precision floating-point number.
  println( float f )    : Prints a floating-point number.
  println( int i )      : Prints an integer.
  println( long l )     : Prints a long integer.
  println( Object obj ) : Prints an object.
  println( String s )   : Prints a string.
  ```

1.6 Exceptions

Cette section revisite quelques concepts liés aux traitements d'exception en Java et présente ceux qui sont spécifiques aux traitements des entrées-sorties.

1.6.1 IOException

"Signals that an I/O exception of some sort has occurred. This class is the general class of exceptions produced by failed or interrupted I/O operations." **IOException** est une sous-classe d'**Exception**. Ces exceptions doivent être traitées : à l'aide de blocs **try/catch** ou d'une déclaration.

1.6.2 FileNotFoundException

Le constructeur **FileInputStream(String name)** déclare une exception de type **FileNotFoundException**.

1.6.3 finally

Les énoncés du bloc finally d'un énoncé **try/catch** sont toujours exécutés. On l'utilise pour des constructions comme celles-ci.

```
Object val = null;
try {
    val = pre();
    // traitements
} finally {
    if ( val != null ) {
       post();
    }
}
```

Ici, lors d'un appel sans erreur à la méthode **pre()**, le résultat est une référence non **null**. Suite à l'appel à **pre()**, certains traitements seront exécutés. Qu'il y ait ou non une erreur, lors de ces traitements, on souhaite effectuer un posttraitement. Dans le bloc **finally**, on sait que l'appel à la méthode **pre()** a été un succès parce que la valeur de **val** est non **null**. On fait alors appel à la méthode **post()**.

Voici une situation typique ou ce mécanisme est nécessaire.

```
public static void copy( String fileName ) throws IOException, FileNotFoundException {

   InputStreamReader input = null;
   try {
       input = new InputStreamReader( new FileInputStream( fileName ) );
       int c;
       while ( ( c = input.read() ) != -1 ) {
           System.out.write( c );
       }
   } finally {
       if ( input != null )
           input.close();
   }
}
```

Tous les systèmes d'exploitation limitent le nombre maximum de fichiers qui peuvent être ouverts simultanément ; sous Linux, cette limite est de 16 fichiers par défaut. Il faut donc toujours s'assurer de fermer tous les fichiers, dès que possible.

La méthode **close** déclare deux exceptions, ainsi la méthode **copy** doit les déclarer (ou les attraper). Voir Copy.java

1.7 Formatage (nombres)

Les solutions proposées pour le formatage de données sont plus complexes que pour certains autres langages. Le langage C a été développé au début des années 1970s, alors que Java a été développé dans les années 1990s. Java propose plusieurs solutions pour l'internationalisation de programmes — utiliser "." ou ","afin de séparer les décimales d'un nombre selon la configuration locale de l'ordinateur utilisé.

1.7.1 java.text.Format

Format est la super-classe (abstraite) des classes liées au formatage : entre autres **DateFormat** et **NumberFormat**. Voyez par vous même ce qui arrive lorsqu'on affiche un nombre en point flottant (Test.java).

```
> java Test
3.141592653589793
```

C'est parfois le résultat désiré et parfois pas (si on souhaite imprimer un montant d'argent, par exemple). Afin d'imprimer un nombre fixe de décimales, nous utiliserons **NumberFormat**. Tout d'abord, il faut importer cette classe. Ensuite, on crée une instance et l'on détermine le nombre de décimales à l'aide des méthodes **setMaximumFractionDigits** et **setMinimumFractionDigits**. Finalement, on utilise la méthode format afin de créer la chaîne désirée. Voir Test.java

```
import java.text.NumberFormat;

public class Test {

    public static void main( String[] args ) {

        NumberFormat nf = NumberFormat.getInstance();
        nf.setMaximumFractionDigits(2);
        nf.setMinimumFractionDigits(2);

        System.out.println( nf.format( 1.0/3.0 ) );
        System.out.println( nf.format( 100.0 ) );
    }
}
```

1.7.2 DecimalFormat

Voici une méthode alternative. Ici, nous affichons trois décimales dans une chaîne de taille 8.

```
(adapté du tutoriel de Jean Vaucher (et Guy Lapalme))

import java.text.DecimalFormat;
import java.text.DecimalFormatSymbols;

float X,Y = .....;
String format1 = "###0.000" ;

static DecimalFormat fm1 = new DecimalFormat( format1,
                                new DecimalFormatSymbols( Locale.US ) );

System.out.println("X=" + fm1.format(x) + ", Y=" + fm1.format(y));
```

Voir Test.java

2 PlayListManager

Pour ce laboratoire, vous devez modifier l'application **PlayListManager** afin de lire et écrire les chansons dans des fichiers.

2.1 Lire les chansons à partir d'un fichier

Modifiez l'application afin de lire les chansons à partir d'un fichier. Par exemple, donnez le nom fichier sur la ligne de commande,

```
> java Run songs.csv
```

Le fichier contient une entrée par ligne. Chaque entrée est composée du titre, du nom de l'artiste et du titre de l'album. Les champs sont séparés par " :".

```
A Dream Within A Dream:Alan Parsons Project:Tales Of Mystery & Imagination
Aerials:System Of A Down:Toxicity
Bullet The Blue Sky:U2:Joshua Tree
Clint Eastwood:Gorillaz:Clint Eastwood
Flood:Jars Of Clay:Jars Of Clay
Goodbye Mr. Ed:Tin Machine:Oy Vey, Baby
Here Comes The Sun:Nina Simone:Anthology
In Repair:Our Lady Peace:Spiritual Machines
In The End:Linkin Park:Hybrid Theory
Is There Anybody Out There?:Pink Floyd:The Wall
Karma Police:Radiohead:OK Computer
Le Deserteur:Vian, Boris:Titres Chansons D'auteurs
Les Bourgeois:Brel, Jacques:Le Plat Pays
Mosh:Eminem:Encore
Mosquito Song:Queens Of The Stone Age:Songs For The Deaf
New Orleans Is Sinking:Tragically Hip, The:Up To Here
Pour un instant:Harmonium:Harmonium
Sweet Dreams:Marilyn Manson:Smells Like Children
Sweet Lullaby:Deep Forest:Essence of the forest
Yellow:Coldplay:Parachutes
```

2.2 Écrire les chansons dans un fichier

Modifiez l'application afin de sauvegarder les chansons de la nouvelle liste dans un fichier. Vous pouvez spécifier le nom du fichier sur la ligne de commande,

```
> java Run songs.csv workout.csv
```

Le format du fichier est le même que celui en entrée.

Solution

Il est fortement suggéré que vous développiez vos solutions des méthodes **PlayList getSongsFromFile(String fileName)** et **void writeSongsToFile(String fileName)** dans une classe séparée, par exemple **Utils**. Une fois le travail terminé, intégrez les méthodes à l'application **PlayListManager**.

- media.jar (starting point)
- media.jar (solution)
- src.jar (all the files for this laboratory)

Ressources

- java.sun.com/docs/books/tutorial/essential/io
- www.iro.umontreal.ca/∼vaucher/Java/tutorials/Java_files.txt

Dernière modification : 23 mars 2007