Les Entrées-Sorties & sérialisation

Les entrées-sorties

- Le package « java.io » permet de gérer la manipulation d'entrées / sorties.
- Les entrées et les sorties sont des flux de données

- flux entrants : le programme peut lire les données.

- flux sortants : le programme peut y émettre des données.

- Source de données

-Un flux connecte un programme avec une sources de données

- Ex. écran/clavier, fichiers, connexion TCP/IP, Web

Les classes abstraites

Classe abstraites

- java.io.InputStream/ java.io.OutputStream : flux entrant/ sortant pour données brutes (octets)

- java.io.Reader/ java.io.Writer : flux entrant/ sortant pour données textes uni codes

Exception

- java.io.IOException : représente des problèmes de la connexion de flux

- Dépendant des sources de données : Fichiers, Web,..

InputStream/ OutputStream

1) Flux sortant OutputStream (Ecriture)

Création d'un fichier binaire

2) Flux Entrant InputStream (Lecture)

Lecture d'un fichier binaire

1-Création séquentielle d'un fichier binaire

- La classe abstraite OutputStream *sert de base à toutes les classes relatives à des flux binaires* de sortie. La classe FileOutputStream, *dérivée de* OutputStream, *permet de manipuler un flux* binaire associé à un fichier en écriture.

```
FileOutputStream f = new FileOutputStream ("entiers.dat") ;
```

Cette opération associe l'objet f *à un fichier de nom* entiers.dat

- Si ce fichier n'existe pas, il est alors créé (vide). S'il existe déjà, son ancien contenu est détruit. On a donc affaire à une classique opération d'ouverture d'un fichier en écriture.

- les méthodes de la classe FileOutputStream *sont rudimentaires. En effet, elles* permettent seulement d'envoyer sur le flux (donc d'écrire dans le fichier) un octet ou un tableau d'octets.

DataOutputStream

- il existe une classe *DataOutputStream qui comporte des* méthodes plus évoluées et qui dispose (entre autres) d'un constructeur recevant en argument un objet de type *FileOutputStream.*

```
DataOutputStream sortie = new DataOutputStream (f) ;
```

- on crée un objet *sortie qui, par l'intermédiaire de l'objet f, se trouve associé au fichier entiers.dat.*

```
( ou bien) DataOutputStream sortie = new DataOutputStream ( new
            FileOutputStream ("entiers.dat)) ;
```

- La classe *DataOutputStream dispose notamment de méthodes permettant d'envoyer sur un* flux (donc ici d'écrire dans un fichier) une valeur d'un type primitif quelconque. Elles se nomment *writeInt (pour int), writeFloat (pour float), et ainsi de suite.*

Exemple de programme

un programme complet qui lit des nombres entiers au clavier et qui les recopie dans un fichier binaire

```
import java.io.* ; // pour les classes flux
public class Crsfic1{
 public static void main (String args[]) throws IOException {
String nomfich ;
int n ;
System.out.print ("donnez le nom du fichier a creer : ") ;
nomfich = Clavier.lireString() ;

DataOutputStream sortie = new DataOutputStream ( new
FileOutputStream (nomfich)) ;

do {
       System.out.print ("donnez un entier : ") ;
          n = Clavier.lireInt() ;
          if (n != 0)
         sortie. writeInt (n) ;
}while (n != 0) ;

sortie.close () ;
System.out.println ("*** fin creation fichier ***");
}
}
```

```
donnez le nom du fichier a
creer : entiers.dat
donnez un entier : 12
donnez un entier : 85
donnez un entier : 55
donnez un entier : 128
donnez un entier : 47
donnez un entier : 0
*** fin creation fichier ***
```

Remarques

- la clause *throws IOException dans l'en-tête main. En* effet, la méthode *writeInt (comme toutes les méthodes d'écriture de DataOutputStrream)* peut déclencher une exception (explicite) du type *IOException en cas d'erreur d'écriture.*

- Java dispose d'une classe *File permettant de manipuler des* noms de fichiers ou de répertoires. On pourrait fournir en argument du constructeur de *FileOutputStream un objet de type File à la place d'un objet de type String.*

- Il est possible de doter un flux d'un *tampon. Il s'agit d'un emplacement mémoire qui* sert à optimiser les échanges avec le flux. Les informations sont d'abord enregistrées dans le tampon, et ce n'est que lorsque ce dernier est plein qu'il est "vidé" dans le flux.

- Pour doter un flux de type *FileOutputStream d'un tampon, on crée un objet de type BufferedOutputStream* en passant le premier en argument de son constructeur.

suite

- Voici comment nous pourrions modifier dans ce sens l'initialisation de la variable sortie du programme précédent :

```
DataOutputStream sortie = new DataOutputStream( new
        BufferedOutputStream ( new FileOutputStream (nomfich))) ;
```

- *Lorsqu'un flux est doté d'un tampon*, sa fermeture (*close*) *vide tout naturellement le* tampon dans le flux. On peut aussi provoquer ce vidage à tout moment en recourant à la méthode **flush**.

2-Lecture séquentielle d'un fichier binaire

- Par analogie avec ce qui précède, la classe abstraite *InputStream* sert de base à toute classe relative à des flux binaires d'entrée. La classe *FileInputStream*, dérivée de *InputStream*, *permet de manipuler un flux binaire associé à un fichier en lecture.*

```
FileInputStream f = new FileInputStream ("entiers.dat") ;
```

- Cette opération associe l'objet *f à un fichier de nom entiers.dat. Si ce fichier n'existe pas, une* exception *FileNotFoundException (dérivée de IOException) est déclenchée.*

- les méthodes de la classe *FileInputStream sont rudimentaires. En effet, elles permettent* seulement de lire dans un fichier un octet ou un tableau d'octets. Ici encore, il existe une classe *DataInputStream qui possède des méthodes plus évoluées et qui dispose (entre* autres) d'un constructeur recevant en argument un objet de type *FileInputStream.*

Suite

- *Avec :* DataInputStream entree = new DataInputStream (f) ;
- on crée un objet *entree qui, par l'intermédiaire de l'objet f, se trouve associé au fichier entiers.dat. Ici encore, les deux instructions (création FileInputStream et création DataInputStream) peuvent être condensées en :*

(ou bien) DataInputStream entree = new DataInputStream

(new FileInputStream ("entiers.dat")) ;

- *la classe DataInputStream dispose de méthodes permettant de lire sur un flux (donc ici dans un fichier) une valeur d'un type primitif quelconque. Elles se nomment readInt (pour int), readFloat (pour float)... Ici, readInt nous conviendra.*

Exemple

```
import javax.swing.*;//
import java.io.*;
public class Lecsfic1{
public static void main (String args[]) throws IOException{
String nomfich ;
int n = 0 ;
System.out.print ("donnez le nom du fichier a lister : ") ;
nomfich = Clavier.lireString() ;

DataInputStream entree = new DataInputStream
( new FileInputStream (nomfich)) ;

System.out.println ("valeurs lues dans le fichier " + nomfich + " :") ;
boolean eof = false ; // sera mis a true par exception EOFile
while (!eof){
            try{
                n = entree.readInt () ;
                }catch (EOFException e){
                                                eof = true ;
                                            }
        if (!eof) System.out.println (n) ;
}
entree.close () ;
System.out.println ("*** fin liste fichier ***");
}
}
```

```
donnez le nom du fichier
a lister : entiers.dat
valeurs lues dans le fichier
entiers.dat :
12
85
55
128
47
*** fin liste fichier ***
```

Remarques

- On retrouve dans *main la clause throws IOException correspondant aux exceptions de type IOException, susceptibles d'être déclenchées par readInt en cas d'erreur.*

- Ici, toutefois, nous avons souhaité pouvoir lire un fichier comportant un nombre quelconque d'entiers. C'est pourquoi nous avons utilisé le canevas de lecture suivant :

```
while (!eof) {    // a l'entree eof est false
                        try{
                                n = entree.readInt () ;
                          } catch (EOFException e) {
                                  eof = true ; // il passera a true lors d'une rencontre de fin de fichier
                          }
if (!eof) System.out.println (n) ;
}
```

- En effet, assez curieusement, la fin de fichier apparaît en Java comme une exception. Cela nous oblige donc à créer un bloc try réduit à une seule instruction de lecture, et à détourner un peu la gestion d'exceptions de son but premier, à savoir gérer des conditions exceptionnelles.

*Flux texte (*Writer /Reader)

1) Flux sortant Writer (Ecriture)

Création d'un fichier texte

2) Flux Entrant Reader (Lecture)

Lecture d'un fichier texte

Classes : Writer et Reader

- Java dispose de deux autres familles de classes, dérivées des classes abstraites *Writer et Reader, permettant de manipuler des flux texte. Comme on s'y attend, les caractères* ainsi manipulés subiront alors une transformation, à savoir :

- pour un flux en sortie : une conversion de deux octets représentant un caractère Unicode en un octet correspondant au code local de ce caractère dans l'implémentation ;

- pour un flux en entrée : une conversion d'un octet représentant un caractère dans le code local en deux octets correspondant au code Unicode de ce caractère.

1-Création d'un fichier texte

- Nous vous proposons d'écrire un programme qui lit des nombres entiers au clavier et qui, pour chacun d'entre eux, écrit une ligne d'un fichier texte contenant le nombre fourni accompagné de son carré, sous la forme suivante : 12 a pour carrre 144

- On convient que l'utilisateur fournira la valeur 0 pour signaler qu'il n'a plus de valeurs à entrer.

- La classe abstraite *Writer sert de base à toutes les classes relatives à un flux texte de sortie. La* classe *FileWriter, dérivée de Writer, permet de manipuler un flux texte associé à un fichier.* L'un de ses constructeurs s'utilise ainsi : FileWriter f = new FileWriter ("carres.txt") ;

- Cette opération associe l'objet *f à un fichier de nom* carres.txt*. S'il n'existe pas, il est créé* (vide). S'il existe, son ancien contenu est détruit. On a donc affaire à un classique ouverture en écriture.

suite

- Si l'on souhaite disposer de possibilités de formatage, on peut recourir à la classe *PrintWriter qui dispose* d'un constructeur recevant en argument un objet de type *FileWriter. Ainsi, avec :*

```
PrintWriter sortie = new PrintWriter (f) ;
```

- on crée un objet *sortie qui, par l'intermédiaire de l'objet f, se trouve associé au fichier carres.txt. Bien entendu, les deux instructions peuvent être fusionnées en :*

```
PrintWriter sortie = new PrintWriter (new FileWriter ("carres.txt")) ;
```

- la classe *PrintWriter dispose des méthodes print* et *println que nous allons pouvoir utiliser exactement comme nous l'aurions fait pour afficher* l'information voulue à l'écran.

Exemple :

```
import java.io.* ;

public class Crftxt1{

public static void main (String args[]) throws IOException{
String nomfich ;
int n ;
System.out.print ("Donnez le nom du fichier a creer : ") ;
nomfich = Clavier.lireString() ;

PrintWriter sortie = new PrintWriter (new FileWriter (nomfich)) ;
do{
          System.out.print ("donnez un entier : ") ;
            n = Clavier.lireInt() ;
                  if (n != 0){
                        sortie.println (n + " a pour carre " + n*n) ;
                          }
     }while (n != 0) ;

sortie.close () ;
System.out.println ("*** fin creation fichier ***");
}
}
```

Donnez le nom du fichier a creer :
carres.txt

donnez un entier :
5
donnez un entier :
12
donnez un entier :
45

Remarques

- un flux texte de sortie peut théoriquement être doté d'un tampon. Par exemple, pour doter d'un tampon un flux de type *PrintWriter, on crée* un objet de type *BufferedWriter en passant le premier en argument de son constructeur :*

```
PrintWriter sortie = new PrintWriter (new BufferedWriter (new FileWriter
("carres.txt"))) ;
```

- Cependant, la classe *PrintWriter dispose déjà de son propre tampon, de sorte que cette* démarche est rarement utile.

2-Lecture d'un fichier texte

- On va voir maintenant comment relire de tels fichiers texte. Cependant, cette fois, il n'existe pas de classe symétrique de *PrintWriter*. *Nous allons voir comment procéder en distinguant deux situations :*

a) On se contente d'accéder aux lignes du fichier (sans chercher à en interpréter le contenu).

b) On souhaite pouvoir accéder aux différentes informations présentes dans une ligne.

Accès aux lignes d'un fichier texte

- il n'existe pas de classe jouant le rôle symétrique de *PrintWriter. Il faut se contenter* de la classe FileReader, *symétrique de* FileWriter. *En la couplant avec la classe BufferedReader qui dispose d'une méthode* readLine, *nous allons pouvoir lire chacune des lignes de* notre fichier (avec FileReader *seule, on ne pourrait accéder qu'à des caractères, et il nous* faudrait prendre en charge la gestion de la fin de ligne). Nous créons donc un objet *entree de* la façon suivante :

```
BufferedReader entree = new BufferedReader (new FileReader ("carres.txt")) ;
```

- La méthode readLine *de la classe BufferedReader fournit une référence à une chaîne correspondant* à une ligne du fichier. Si la fin de fichier a été atteinte avant que la lecture ait commencé, autrement dit si aucun caractère n'est disponible (pas même une simple fin de ligne), readLine *fournit la valeur* null. *Il est donc possible de parcourir les différentes lignes de notre* fichier, sans avoir besoin cette fois de recourir à la gestion des exceptions. Il suffit d'employer un des canevas suivants :

```
do{
        ligne = entree.readLine() ;
      if (ligne != null) { // traitement d' une ligne }
}while (ligne != null) ;

while (true){
          ligne = entree.readLine() ;
                    if (ligne != null) break ;
                    // traitement d' une ligne
            }
```

Exemple

```
import java.io.* ;
public class Lecftxt1{
public static void main (String args[]) throws IOException{

String nomfich ;
String ligne ;

System.out.print ("Donnez le nom du fichier a lister : ") ;
nomfich = Clavier.lireString() ;

BufferedReader entree = new BufferedReader (new FileReader
(nomfich)) ;
do {
        ligne=entree.readLine();
          if(ligne≠null) System.out.println(ligne);
    }while(ligne≠null);

entree.closee();
System.out.println("****fin liste fichier****");
}
}
```

```
Donnez le nom du
fichier a lister :
carres.txt
5 a pour carre 25
12 a pour carre 144
45 a pour carre 2025
2 a pour carre 4
*** fin liste fichier ***
```

La classe StringTokenizer

- Dans certains cas, vous pourrez avoir besoin d'accéder à chacune des informations d'une même ligne. Nous l'avons déjà dit, Java ne dispose pas de fonctionnalités de lecture formatée analogues à celles d'écriture formatée que procure la classe PrintWriter. *En revanche,* il dispose d'une classe utilitaire nommée StringTokenizer*, qui permet de découper une* chaîne en différents tokens *(sous-chaînes), en se fondant sur la présence de caractères* séparateurs qu'on choisit librement. Par ailleurs, on pourra appliquer à ces différents tokens*, les* possibilités de conversion d'une chaîne en un type primitif, ce qui permettra d'obtenir les valeurs voulues.

La gestion des fichiers : la classe File

- Java dispose d'une classe *File qui offre des fonctionnalités de gestion de fichiers comparables* à celles auxquelles on accède par le biais de commandes système de l'environnement
- Il s'agit d'opérations concernant le fichier dans sa globalité, et non plus chacune des informations qu'il contient.
- Création d'un objet de type File :

```
File monFichier = new File ("truc.dat") ;
```

crée un objet de type *File, nommé* monFichier*, auquel est associé le nom truc.dat.*

- Malgré leur nom, les objets de type *File peuvent être associés, non seulement à un nom de* fichier, mais aussi à un nom de répertoire.

Utilisation d'objets de type File

- **Dans les constructeurs de flux :**
- Par exemple, au lieu de :

```
DataOutputStream sortie = new DataOutputStream
( new FileOutputStream ("entiers.dat)) ;
```

- vous pourriez utiliser :

```
File objFich = new File ("entiers.dat") ;
DataOutputStream sortie = new DataOutputStream
  ( new FileOutputStream (objFich)) ;
```

Les flux en général

- Les différentes classes de flux peuvent se répartir en familles, chacune issue d'une classe de base abstraite :
- • *OutputStream : flux binaires de sortie ;*
- • *InputStream : flux binaires d'entrée ;*
- • *Writer : flux texte de sortie ;*
- • *Reader : flux texte d'entrée.*

La sérialisation

Sérialisation

- *Permet de rendre des objets persistants en écrivant les données présentes en mémoire vers un flux de données binaires*
- *introduction dans le JDK 1.1 d'un mécanisme de sérialisation*

  - *permet de sérialiser les objets de manière transparente et indépendante du système d'exploitation.*

  - *s'appuie sur les flux d'entrée sortie* ***(java.io)***

- Implémente le mécanisme de sérialisation
- **ObjectOutputStream** *herite de* **OutputStream**
- Implémente le mécanisme de désérialisation
- **ObjectInputStream** *herite de* **InputStream**
- **interface Serializable** : Permet d'identifier les classes sérialisables

Interface Serializable

- Pour qu'un objet puisse être sérialisé sa classe implémenter l'interface **Serializable** ou hériter d'une classe elle-même sérialisable.
- La sérialisation d'un objet consiste à écrire ses attributs sur un flux de sortie binaire.
- Tout attribut est sérialisé si :
- *Il est de type primitif (int, char, boolean, ….)*
- *référence dont le type est un type sérialisable.*
- *Il n'est pas déclaré static*
- *Il n'est pas déclaré transient*
- *Il n'est pas hérité d'une classe mère sauf si celle-ci est elle-même*
- *sérialisable*

Classes ObjectOutputStream et ObjectInputStream

- **ObjectOutputStream** représente "un flux objet" qui permet de sérialiser un objet grâce à la méthode **writeObject()**.
- **ObjectInputStream** représente "un flux objet" qui permet de désérialiser un objet grâce à la méthode **Object readObject()**.

Le serialVersionUID

- Le serialVersionUID est un "numéro de version", associé à toute classe implémentant l'interface *Serializable.*
- *Permet de s'assurer, lors de la désérialisation, que les versions des classes Java soient concordantes.*
- *Si le test échoue, une InvalidClassException est levée.*

```
private static final long serialVersionUID = 42L;
```

```java
import java.io.Serializable;

public class Personne implements Serializable {
 static private final long serialVersionUID = 6L;
 private String nom;
 private String prenom;
 private Integer age;

 public Personne(String nom, String prenom, Integer age) {
  this.nom = nom;
  this.prenom = prenom;
  this.age = age;
 }

 public String toString() {
  return nom + " " + nom + " " + age + " ans";
 }
}
```

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

public class SerializationMain {

 static public void main(String ...args) {
  try {
   // création d'une personne
   Personne p = new Personne("Dupont", "Jean", 36);
   System.out.println("creation de : " + p);

   // ouverture d'un flux de sortie vers le fichier "personne.serial"
   FileOutputStream fos = new FileOutputStream("personne.serial");

   // création d'un "flux objet" avec le flux fichier
   ObjectOutputStream oos= new ObjectOutputStream(fos);
   try {
    // sérialisation : écriture de l'objet dans le flux de sortie
    oos.writeObject(p);
    // on vide le tampon
    oos.flush();
    System.out.println(p + " a ete serialise");
   } finally {
    //fermeture des flux
    try {
     oos.close();
    } finally {
     fos.close();
    }
   }
  } catch(IOException ioe) {
   ioe.printStackTrace();
  }
 }
}
```

```
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

public class DeserializationMain {

 static public void main(String ...args) {
  Personne p = null;
  try {
   // ouverture d'un flux d'entrée depuis le fichier "personne.serial"
   FileInputStream fis = new FileInputStream("personne.serial");
   // création d'un "flux objet" avec le flux fichier
   ObjectInputStream ois= new ObjectInputStream(fis);
   try {
    // désérialisation : lecture de l'objet depuis le flux d'entrée
    p = (Personne) ois.readObject();
   } finally {
    // on ferme les flux
    try {
     ois.close();
    } finally {
     fis.close();
    }
   }
  } catch(IOException ioe) {
   ioe.printStackTrace();
  } catch(ClassNotFoundException cnfe) {
   cnfe.printStackTrace();
  }
  if(p != null) {
   System.out.println(p + " a ete deserialise");
  }
 }
}
```

Le mot clé transient

- Le mot clé **transient** **permet d'interdire la sérialisation d'un attribut d'une classe**.
- en général utilisé pour les données sensibles =’ ’ mots de passe’ ’
- Pas utile à sérialiser.

- **Question** : reprendre l’ exemple précédent et ajouter un champs **Password** avec transient : sérialiser et deserieliser.