Entrées / Sorties en Java

Les entrées/sorties sont gérées par des classes dont la plupart sont dans le paquetage `java.io`. Ces classes permettent de traiter des flux (*stream*) de données. Un flux représente une suite d'octets (par exemple un fichier ou une chaîne de caractères) qu'on peut lire (entrée) ou dans lequel on peut écrire (sortie).

Il existe essentiellement 4 classes "mères" abstraites pour traiter les flux de données :

- `java.io.Reader` et `java.io.Writer` pour les flux de caractères ;
- `java.io.InputStream` et `java.io.OutputStream` pour les flux d'octets ou flux binaires.

Parmi les classes qui héritent de ces classes mères, on trouve :

1. des classes de base ou d'extraction associées à une source ou une destination concrète, par exemple `java.io.FileReader` pour lire un fichier ;
2. des classes de fonctionnalités ou de décoration qui ajoutent des fonctionnalités aux classes d'extraction (ex : utilisation d'un *buffer*, lecture ligne par ligne…).

En composant ces différentes classes, on obtient de multiples fonctionnalités. Par exemple, pour lire un fichier en utilisant un buffer :

```
FileReader r = new FileReader(fichier);
BufferedReader br = new BufferedReader(r);
```

Une fois ouvert, un flux doit être fermé grâce à la méthode `close()`. Ici, appeler `br.close()` appelle automatiquement `r.close()`.

Try-with-resources

Les méthodes de lecture et écriture de fichiers, textes ou binaires, lèvent des exceptions. Afin que l'exception soit gérée et que le flux soit fermé correctement, il existe deux méthodes de gestion des exceptions : la première est l'utilisation d'un bloc **`finally`** :

```
FileReader r = null;
BufferedReader br = null;
try {
     r = new FileReader(fichier);
     br = new BufferedReader(r);
     br.readLine();
} finally {
     if(br != null) {
          br.close();
     }
}
```

Depuis Java 7, une autre méthode existe, c'est l'utilisation d'un **`try`**-with-resources : c'est un bloc **`try`** pour lequel on va associer une ressource. La ressource sera fermée automatiquement lorsque le programme aura fini de l'utiliser. Cette ressource doit donc pouvoir être fermée automatiquement en implémentant l'interface `java.lang.AutoCloseable`.

```
try (
     FileReader r = new FileReader(fichier);
     BufferedReader br = new BufferedReader(r)
) {
     br.readLine();
}
```

Les éventuels blocs **`catch`** et **`finally`** d'un **`try`**-with-resources sont appelés après que la ou les ressources liées aient été fermées.

Fichiers texte

Pour la manipulation des flux textuels, Java dispose de deux familles de classes dérivées des classes abstraites **`Reader`** et **`Writer`** pour respectivement lire et écrire dans un flux.

Lecture d'un ficher texte

Dans la classe d'extraction `java.io.FileReader`, le constructeur `FileReader(`**`String`**` fileNname)` ouvre un flux de lecture à partir du fichier indiqué. Si ce fichier n'existe pas ou que le fichier n'est pas valide (c'est un répertoire par exemple), une exception de type `FileNotFoundException` est levée.

La classe de fonctionnalité `java.io.BufferedReader` lit un flux de caractères en les plaçant dans un tampon, ce qui d'une part permet d'éviter de charger d'un seul coup l'intégralité du fichier en mémoire et d'autre part d'accéder directement aux données du fichier à chaque lecture d'un caractère. Le constructeur **`BufferedReader`**`(Reader in)` ouvre un flux de lecture bufferisé à partir du flux d'entrée `in`, qui peut provenir du clavier, d'un fichier… Elle possède des méthodes de lecture d'un caractère ou d'une ligne.

```
import java.io.*;

public class Lire {
     public static void main(String[] args) throws IOException {
          BufferedReader in = null;
          try {
               // ouverture du fichier
               in = new BufferedReader(
                         new FileReader("essai.txt")
               );
               // lecture des données
               String ligne = null;
               while ((ligne=in.readLine()) != null) {
                    System.out.println(ligne);
               }
          } finally {
               // fermeture du fichier
               if (in != null)
                    in.close();
          }
     }
}
```

La classe d'exception `IOException` permet de signaler qu'un problème d'entrée/sortie est survenu lors de l'ouverture, de la manipulation ou de la fermeture d'un flux.

Depuis Java 5, la classe `java.util.Scanner` permet de lire directement des données primitives :

```
import java.io.*;
import java.util.Scanner;

public class Lire {
     public static void main(String[] args) throws IOException {
          Scanner in = new Scanner(new File("essai.txt"));
          while (in.hasNext()) {
               // lecture des données
               String s = in.next());
               String s = in.nextLine());     // ligne entière
               if (in.hasNextInt()) // si la donnée suivante
                    int i = in.nextInt();     // est un entier
               double d = in.nextDouble();
          }
          in.close();
     }
}
```

Cette classe dispose aussi de méthodes `hasNextXXX()` permettant de vérifier le type du prochain token.

Écriture dans un ficher texte

Dans la classe d'extraction `java.io.FileWriter`, le constructeur `FileWriter(`**`String`**` fileName)` permet d'écrire dans le fichier de nom `fileName`. S'il n'existe pas, il est créé automatiquement.

La classe de fonctionnalité `java.io.PrintWriter` facilite l'écriture de différents types de données dans un flux de caractères, en fonction de leur type. Elle dispose d'un constructeur recevant en argument un objet de type `Writer`. La classe `PrintWriter` possède les méthodes `print(T)` et `println(T)` qui, à la manière de `Scanner`, permettent d'écrire des données primitives dans un flux de caractères.

```java
import java.io.*;

public class Ecrire {
     public static void main(String[] args) throws IOException {
          PrintWriter out = null;
          try {
               // ouverture du fichier
               out = new PrintWriter(
                         new FileWriter("Resultats.txt")
               );
               // écriture dans le fichier
               out.println("Dubois");
               out.println(42);
               out.println(3.14);
          } finally {
               // fermeture du flux
               if (out != null)
                    out.close();
          }
     }
}
```

Écriture à l'écran et lecture depuis le clavier

System.*out* permet des écritures vers la sortie standard. **System.***in* est généralement associée à un `BufferedReader` pour pouvoir lire facilement grâce à la méthode `readLine()`.

```java
import java.io.*;

public class Clavier {
     public static void main(String[] args) throws IOException {
          BufferedReader in;
          in = new BufferedReader(new InputStreamReader(System.in));
          String ligne = null;
          while ((ligne=in.readLine()) != null) {
               System.out.println(ligne);
          }
     }
}
```

Fichiers binaires

Lire un fichier binaire

La classe `java.io.FileInputStream` est une classe d'extraction qui permet de construire un flux d'octets d'un programme vers un fichier. La classe `java.io.DataInputStream` permet de lire les données primitives d'un objet dans un flux de données binaires. La classe `java.io.ObjectInputStream` permet de lire directement l'objet dans le flux, avec ses champs. Cependant, un tel objet doit implémenter l'interface `java.io.Serializable`. Cette interface ne nécessite pas de redéfinir une quelconque méthode mais il est nécessaire de définir un attribut `serialVersionUID` permettant d'identifier la classe. Eclipse peut le générer aléatoirement.

```
import java.io.*;

class Personne implements Serializable {
     private static final long serialVersionUID = 861500525496309628L;
     …
}

public class LireBinaire {
     public static void main(String[] args) throws IOException {
          ObjectInputStream ois;
          ois = new ObjectInputStream(
                    new FileInputStream("objets")
          );
          try {
               while (true) {
                    System.out.println(ois.readObject().toString());
               }
          } catch (ClassNotFoundException cnfe) {
               // erreur de lecture
          } catch (EOFException eofe) {
               // fin de fichier
          }
          ois.close();
     }
}
```

Écriture dans un fichier binaire

La classe `java.io.FileOutputStream` est une classe d'extraction qui permet de construire un flux d'octets du programme vers un fichier. La classe `java.io.DataOutputStream` permet d'écrire les données primitives d'un objet dans un flux. La classe `java.io.ObjectOutputStream` permet d'écrire directement l'objet dans le flux, avec ses champs.

```
import java.io.*;

public class EcrireBinaire {
     public static void main(String[] args) throws IOException {
          ObjectOutputStream oos;
          oos = new ObjectOutputStream(
                    new FileOutputStream("objets")
          );
          oos.writeObject(new Personne(…));
          oos.close();
     }
}
```

La classe `Personne` implémente l'interface `Serializable`. Il existe deux méthodes pour écrire un objet `o` dans le flux : `writeObject(o)` et `writeUnshared(o)`. La différence entre les deux est que `writeObject(o)` écrit les objets de manière unique. Autrement dit, un même objet[1] ne sera inscrit qu'une seule fois dans le flux de données ; `writeUnshared(o)`, elle, écrit les objets dans le flux qu'ils aient été écrits précédemment ou non.

À noter que ces méthodes ne fonctionnent que si les champs des objets sont des types primitifs. Si la conversion d'un objet en données binaires n'est pas trivial, il est nécessaire de définir les méthodes `writeObject(`**`ObjectOutputStream`**`)` et `readObject(`**`ObjectInputStream`**`)`[2].

[1] « même objet » au sens où `o1 == o2` et non `o1.equals(o2)`
[2] consulter la documentation de `java.io.Serializable` pour plus d'informations