Support de cours sur les Entrées/Sorties en Java

Généralités

Il y a **quatre** classes "mères", abstraites, pour traiter les flots de données héritant directement d'`Object` :

pour traiter des flots d'octets (**fichiers binaire**)

la classe **`InputStream`**

la classe **`OutputStream`**

pour traiter des flots de caractères (**fichiers textes**)

la classe **`Reader`**

la classe **`Writer`**

Beaucoup de classes héritent de ces classes de base. Nous ne traiterons que de celles qui servent dans les applications les plus courantes.

On peut envelopper en cascade les streams/readers/writers pour disposer des fonctionnalités nécessaires. Exemple les InputStreams et les DataInputStreams :

Lire des lignes avec un `BuffereReader`

```
import java.io.*;

class SaisieClavier
{
  public static void main (String[] argv) throws IOException
  {
    String ligne;
    BufferedReader entree = new BufferedReader (new
InputStreamReader(System.in));

     ligne = entree.readLine();

    while(ligne.length() > 0)
    {
        System.out.prinltn(ligne);
        ligne = entree.readLine();
     }
  }
```

}

On utilise deux classes de `java.io`, la classe `InputStreamReader` et la classe `BufferedReader`.

La classe `InputStreamReader` admet un constructeur `InputStreamReader(InputStream)`, c'est-à-dire un constructeur qui admet en paramètre un flot d'entrée.
`System.in` est une instance de la classe `InputStream`.
Avec une instance de `InputStreamReader`, **on ne peut grosso modo que lire des caractères un à un**.

La classe `BufferedReader` a un constructeur qui prend en argument une instance de `Reader` dont hérite la classe `InputStreamReader`. Cette classe **permet en particulier de lire une ligne d'un texte**, mais en revanche, on ne peut pas lui demander de lire un entier, un double etc...

Lire des entiers, des réels, des mots avec un `StreamTokenizer`

StreamTokenizer entree= new StreamTokenizer (new InputStreamReader(System.in));

```
boolean fin = false;
int type;

while (! fin) {
        type=entree.nextToken();
        switch (type) {
                case StreamTokenizer.TT_NUMBER:
                        System.out.println("Nombre : "+entree.nval);
                        break;
                case StreamTokenizer.TT_WORD:
                        System.out.println("Mot : "+entree.sval);
                        break;
                 default:
                        fin = true;
        }
}
```

On utilise ici une instance de `StreamTokenizer` qui est un analyseur syntaxique plutôt rudimentaire.
Un constructeur de la classe `StreamTokenizer` prend en paramêtre une instance de `Reader`.

La méthode `nextToken()` de la classe `StreamTokenizer` retourne le type de l'unité lexicale suivante, type qui est caractérisé par une constante entière. Cela peut être :

- `TT_NUMBER` si l'unité lexicale représente un nombre. Ce nombre se trouve alors dans le champ `nval` de l'instance de `StreamTokenizer`. Ce champ est de type **`double`**.
- `TT_WORD` si l'unité lexicale représente une chaîne de caractères. Cette chaîne se trouve alors dans le champ `sval` de l'instance du `StreamTokenizer`.
- `TT_EOL` si l'unité lexicale représente une fin de ligne. La fin de ligne n'est reconnue comme unité lexicale que si on a utilisé l'instruction

```
nomDuStreamTokenizer.eolIsSignificant(true);
```

- `TT_EOF` s'il s'agit du signe de fin de fichier.

Lire/écrire des caractères dans un fichier

```
FileReader lecteur;
FileWriter ecrivain;
int c;

lecteur = new FileReader("essai.txt");
ecrivain = new FileWriter("copie_essai.txt");
ecrivain.write("copie de essai.txt\n");
while((c = lecteur.read()) != -1)
  ecrivain.write(c);
lecteur.close();
ecrivain.close();
```

Si le fichier `essai.txt` contient :

```
bonjour
rebonjour
```

le fichier copie_essai.txt contient après exécution :

```
copie de essai.txt
bonjour
rebonjour
```

La classe `FileReader` permet de lire des caractères dans un fichier.
La classe `FileWriter` permet d'écrire des caractères dans un fichier

Analogie avec fgetc et fputc du C

Ecrire diverses choses dans un fichier texte

On utilise ici une instance de `PrintWriter`, dont un constructeur prend en argument un `Writer` dont la classe `BufferedWriter` hérite.

Vous pouvez utiliser avec une instance de `PrintWriter` les méthodes `print` et `println` de la même façon qu'avec `System.out` (qui est de la classe `PrintStream`).
La classe `PrintWriter` (qui hérite de la classe `Writer`) ne fait qu'améliorer la classe `PrintStream` (qui hérite de `OutputStream`).

On aurait pu plus simplement initialiser `ecrivain` par :

```
ecrivain = new PrintWriter(new FileWriter(argv[0]));
```

mais alors les écritures n'utiliseraient pas de **mémoire-tampon**.

```
PrintWriter ecrivain;
int n = 5;

ecrivain =  new PrintWriter(new BufferedWriter
    (new FileWriter(argv[0])));

ecrivain.println("bonjour, comment cela va-t-il ?");
ecrivain.println("un peu difficile ?");
ecrivain.println("On peut mettre des entiers : "+n);
ecrivain.print("On peut mettre des instances de Object : ");
ecrivain.println(new Integer(36));
ecrivain.close();
```

Apres exécution, le fichier indiqué contient :

```
bonjour, comment cela va-t-il ?
un peu difficile ?
On peut mettre des entiers 5
On peut mettre des instances de Object : 36
```

Lire dans un fichier texte

Ligne à ligne

On compose un `BufferdReader` avec un `FileReader`.

```
BufferedReader lecteurAvecBuffer = null;
String ligne;

try
   {
      lecteurAvecBuffer = new BufferedReader
        (new FileReader(argv[0]));
   }
catch(FileNotFoundException exc)
   {
        System.out.println("Erreur d'ouverture");
   }

while ((ligne = lecteurAvecBuffer.readLine()) != null)
     System.out.println(ligne);
lecteurAvecBuffer.close();
```

Avec un StreamTokenizer

Il s'agit maintenant de lire des entiers dans un fichier ne contenant que des entiers et d'en faire la somme.

```
import java.io.*;

class LireEntiers
{
  public static void main (String[] argv) throws IOException
  {
    int somme = 0;
    FileReader fichier = new FileReader(argv[0]);

  StreamTokenizer entree = new StreamTokenizer(fichier);
  while(entree.nextToken() == StreamTokenizer.TT_NUMBER)
   {
            somme += (int)entree.nval;
   }
  System.out.println("La somme vaut : " + somme);
  fichier.close();
 }
}
```

Avec un fichier contenant :

```
5 3 6 2 7
-10 23
```

on obtient :

```
La somme vaut : 36
```

Écrire dans un fichier binaire

Pour traiter des fichiers binaires, c'est-à-dire qui contienne éventuellement autres choses que des caractères (des `int`(s) écrits en binaire et pas comme des chaînes de caractères...), on peut utiliser la classe `DataOutputStream`. Celle-ci à un constructeur qui prend en paramètre une instance de la classe `OutputStream`. On aurait pû initialiser notre variable ecrivain par :

```
DataOutputStream(new FileOutputStream(argv[0]));
```

Cela aurait été identique en apparence, mais si on compose comme il est fait ici avec une instance de `BufferedOutputStream`, les écritures dans le fichier utilisent une mémoire-tampon, ce qui gagne du temps.

La classe `OutputStream` dispose de beaucoup de méthodes ; nous donnons des exemples ci-dessous des principales méthodes.

```
import java.io.*;

class EcrireFichierBinaire
{
  public static void main(String[] argv) throws IOException
  {
    DataOutputStream ecrivain;

    ecrivain =
   new DataOutputStream(new BufferedOutputStream
                                  (new FileOutputStream(argv[0])));

  ecrivain.writeUTF("bonjour");
  ecrivain.writeInt(3);
  ecrivain.writeLong(100000);
  ecrivain.writeFloat((float)2.0);
  ecrivain.writeDouble(3.5);
  ecrivain.writeChar('a');
  ecrivain.writeBoolean(false);
  ecrivain.writeUTF("au revoir");
  System.out.println(ecrivain.size());
  ecrivain.close();
 }
}
```

Après l'exécution de la commande

```
java EcrireFichierBinaire essai
```

on obtient à lécran : 47

et le fichier `essai` est assez illisible.

Lire dans un fichier binaire

La classe DataInputStream nous permet de lire un fichier binaire. D'autre méthodes de cette classe pourront être consultées.
Si on ne désire pas utiliser de mémoire-tampon, on peut initialiser la variable lecteur plus simplement par :

```
lecteur= new DataInputStream(new FileInputStream(argv[0]));
```

```
import java.io.*;

class LireFichierBinaire
{
  public static void main(String[] argv) throws IOException
  {
    DataInputStream lecteur;

    lecteur=
   new DataInputStream(new BufferedInputStream
                              (new FileInputStream(argv[0])));
  System.out.println(lecteur.readUTF());
  System.out.println(lecteur.readInt());
  System.out.println(lecteur.readLong());
  System.out.println(lecteur.readFloat());
  System.out.println(lecteur.readDouble());
  System.out.println(lecteur.readChar());
  System.out.println(lecteur.readBoolean());
  System.out.println(lecteur.readUTF());
  lecteur.close();
 }
}
```

Pour la commande :

```
java LireFichierBinaire essai
```

où le fichier `essai` est issu de l'exécution du programme de l'exemple précédent, on obtient :

```
bonjour
3
100000
2.0
3.5
a
false
au revoir
```

la classe `File`

On se propose ici d'exploiter la classe `java.io.File`. Celle-ci permet de lister les fichiers d'un répertoire, de savoir si un fichier existe, de renommer un fichier, de supprimer un fichier... Une partie des méthodes de la classe `java.io.File` sont illustrées ci-dessous.
Pour exécuter notre programme, on doit indiquer sur la ligne de commande le nom d'un répertoire. On indiquera en fait le répertoire dans lequel s'exécute notre programme de façon à vérifier que le fichier qui sera créé au cours du programme figurera bien ensuite dans ce répertoire.

```
import java.io.*;

class EssaiFile
{
  public static void main(String[] argv) throws IOException
  {
    File  repertoire;
    File fichier=null;
    File nouveauFichier;
    String[] listeFichiers;
    PrintWriter ecrivain;

    repertoire=new File(argv[0]);
  if (!repertoire.isDirectory()) System.exit(0);
  fichier=new File("fichier.essai");
  System.out.println("le fichier "+fichier.getName()+
                (fichier.exists()?" existe":" n'existe pas"));
        //en sortie : le fichier fichier.essai n'existe pas
  ecrivain=new PrintWriter(new FileOutputStream("fichier.essai"));
  ecrivain.println("bonjour");
  ecrivain.close();
  System.out.println("le fichier "+fichier.getName()+
                (fichier.exists()?" existe":" n'existe pas"));
        //en sortie : le fichier fichier.essai existe
  System.out.println("Sa longueur est "+fichier.length());
        //en sortie : Sa longueur est 8
  System.out.println("Son chemin complet est  \n  "+fichier.getAbsolutePath());;
        //en sortie :
//Son chemin complet est
//  /inf/aquilon/infmd/charon/public_html/coursJava/fichiersEtSaisies/fichier.essai
  System.out.println();

  listeFichiers=repertoire.list();
  for (int i=0;i < listeFichiers.length;i++)
              System.out.println(listeFichiers[i]);
  System.out.println();

  nouveauFichier=new File("autre.essai");
  fichier.renameTo(nouveauFichier);
  System.out.println("le fichier "+fichier.getName()+
                (fichier.exists()?" existe":" n'existe plus"));
        //en sortie : le fichier fichier.essai n'existe plus
  System.out.println("le fichier "+nouveauFichier.getName()+
                (nouveauFichier.exists()?" existe":" n'existe pas"));
        //en sortie : le fichier autre.essai existe
  nouveauFichier.delete();
 }
```

}

la sérialisation

La sérialisation consiste à transformer des objets en flots d'octets (l'opération inverse étant appelée dé-sérialisation) qui peuvent être :

- écrits dans des fichiers (on sauve ainsi un objet) et lus depuis des fichiers (on restaure ainsi un objet précédemment sauvé).
- envoyés à travers un réseau (en utilisant par exemple des sockets ou des mécanismes de plus haut niveau comme RMI ou CORBA)

En java pour rendre les objets d'une classe sérialisables il suffit que la classe implémente l'interface java.io.Serializable et que les attributs de cette classe soient eux-mêmes sérialisables (note : les attributs de type scalaire sont automatiquement sérialisables ainsi que la plupart des classes de l'API java) :

```
class Personne implements java.io.Serializable {
        private String nom ;
        private String prenom ;
        private int age ;
        public Personne(String prenom, String nom, int age) {
                this.nom = nom;
                this.prenom = prenom ;
                this.age = age ;
        }
        …
}
```

Dans quelques cas particuliers (listes doublement chaînées par exemple) il faudra développer deux méthodes pour indiquer comment doit se faire la sérialisation et la dé-sérialisation :

```
private void writeObject(java.io.ObjectOutputStream out)
    throws IOException
private void readObject(java.io.ObjectInputStream in)
    throws IOException, ClassNotFoundException;
```

Ces méthodes utiliseront les méthodes des classes `ObjectInputStream` et `ObjectOutputStream` pour lire et écrire les attributs.

Ensuite, pour écrire des objets sur un flot (et dans un fichier) il faudra utiliser la classe `java.io.ObjectOutputStream` de la façon suivante :

```
FileOutputStream fos = new FileOutputStream("fichier.bin");
ObjectOutputStream oos = new ObjectOutputStream(fos);

oos.writeObject("Chaîne de caractère");
oos.writeObject(new Date());
oos.writeObject(new Personne("Bob","Sinclar",30));
oos.close();
```

Et pour les lire (note : il faut utiliser le même ordre, sinon on aura une exception ClassCastException lors du « cast ») `java.io.ObjectInputStream` :

```
FileInputStream fis = new FileInputStream("fichier.bin");
ObjectInputStream ois = new ObjectInputStream(fis);
```

```
String today = (String) ois.readObject();
Date date = (Date) ois.readObject();
Personne p = (Personne) ois.readObject();
ois.close();
```