Nancy-Université

ESSTIN – Semestre 2 – 2006-2007 – Informatique – TD5

Lecture / écriture de fichiers

Dans ce TD, nous allons voir comment écrire / lire un fichier en Java. Pour cela, nous allons utiliser les outils du compilateur, à savoir les outils de gestion de flux de données (*stream*) du paquet `java.io`.

Il existe différents types de flux, dont les principaux sont ceux qui manipulent des octets, et ceux qui manipulent des caractères (codés sur 2 octets en Java). Attention, toutes les méthodes devront comporter `... throws IOException ...` dans leur signature.

Les flux de caractères en lecture (`FileReader`)

Pour lire un fichier, caractère par caractère, nous pouvons utiliser la classe `FileReader`, dont les principaux constructeurs sont :

```
public FileReader(File   file) throws IOException
public FileReader(String path) throws IOException
```

Une fois le fichier ouvert en lecture, nous pouvons récupérer le caractère suivant (sous forme d'entier) via :

```
public int read() throws IOException
```

A l'issue du traitement, nous fermons le fichier via :

```
public void close() throws IOException
```

Les flux de caractères en écriture (`FileWriter`)

Pour écrire des caractères dans un fichier, nous utilions la classe `FileWriter`, dont les principaux constructeurs sont :

```
public FileWriter(File   file) throws IOException
public FileWriter(String path) throws IOException
```

Une fois le fichier ouvert en écriture, nous pouvons y écrire un caractère à la suite du fichier via :

```
public void write(String s) throws IOException
```

A l'issue du traitement, nous fermons le fichier via :

```
public void close() throws IOException
```

Les flux d'octets en lecture (`InputStream`)

Ces flux permettent de lire séquentiellement les informations contenues dans un fichier, octet par octet.

On utilise pour cela la classe `FileInputStream`, ayant pour constructeurs :

```
public FileInputStream (String path)
          throws SecurityException, FileNotFoundException
public FileInputStream (File file)
          throws SecurityException, FileNotFoundException
```

Ces constructeurs permettent de créer une instance de classe `FileInputStream` à partir du chemin d'accès à un fichier path ou d'une instance de classe `File`.

La méthode suivante permet de lire un octet (sous forme d'entier) :
`public int read () throws IOException`

La méthode suivante permet de fermer le fichier :
`public void close () throws IOException`

Les flux d'octets en écriture (`OutputStream`)

Pour écrire des octets dans un fichier, on utilise la classe `FileOutputStream`, ayant pour constructeurs :

```
public FileOutputStream (String path)
          throws SecurityException, FileNotFoundException
public FileOutputStream (File file)
          throws SecurityException, FileNotFoundException
```

Ces constructeurs permettent de créer une instance de classe `FileOutputStream` à partir du chemin d'accès à un fichier path ou d'une instance de classe `File`. Si le fichier n'existe pas il est créé.

La méthode suivante permet d'écrire un octet dans le fichier :
`public void write (int b) throws IOException`

La méthode suivante permet de fermer le fichier :
`public void close () throws IOException`

Exemple d'utilisation # 1

```
import java.io.*;
class FichierCaracteres {
    public static void main(String args[]) throws IOException {
        File f = new File("poeme.txt");
        // le fichier poeme.txt doit exister
        // dans le meme repertoire que vos fichiers java !
        FileReader lecteur = new FileReader(f);
        int car;
        while((car = lecteur.read()) != -1){
            System.out.print ((char)car);
        }
        lecteur.close();

        File g = new File("poeme2.txt");
        FileWriter ecrivain = new FileWriter(g);
        ecrivain.write("la ou tout n'est qu'ordre et beaute");
        ecrivain.close();
    }
}
```

Exemple d'utilisation # 2

```
import java.io.*;
public class FichierOctets {

    public static void main(String[] args) throws IOException {
        FileOutputStream out = new FileOutputStream("test.txt");
        out.write(65);
        out.write(66);
        out.close();
        // verifiez ce que contient le fichier test.txt !

        FileInputStream in = new FileInputStream("test.txt");
        int c;
        while ( (c=in.read()) != -1 ) {
          System.out.println(c);
        }
        in.close();
   }
}
```

Questions

1. Testez les deux exemples ci-dessus et commentez les. Que font-ils ? Quelles classes utilisent-ils ? (consultez la documentation de ces classes dans l'API Java)

2. Concevez un programme qui lit un fichier caractère par caractère, stocke le contenu dans un tableau de caractères (de taille maximale 100), et écrit ce contenu dans un fichier de sortie (vous avez le choix des noms des fichiers).

3. Développez une classe `LecteurOctet` permettant de lire un fichier bit par bit (NB un octet est composé de 8 bits). Pour cela, vous pouvez vous inspirez des informations ci-dessous :

   La méthode `int read()` de `FileInputStream` vous permet de lire un octet (suite de 8 bits valant chacun 0 ou 1, exemple : `00000010` $= 0 \times 2^7 + \ldots + 1 \times 2^1 + 0 \times 2^0$). Il vous faut stocker cet octet, le conserver pour affichage du 1^{er} bit, du 2^{nd}, etc. jusqu'au 8^{e}.
   Lorsque les 8 bits de l'octet on été lus, on récupère un nouvel octet du flot.
   Pour déterminer quel bit doit être pris en compte (en effet, on ne peut lire les données que par octet, *i.e.* par bloc de 8 bits), on utilise un masque (un octet permettant d'isoler le bit à lire, un peu comme un pochoir). Nous notons que ce masque s'applique via l'opérateur *et* (noté `&` en java).
   Au départ le masque est l'entier dont le dernier octet commence par un 1 suivi de zéros : 10000000, soit 128 (2^7), dont le code hexadécimal est 0x80 ('0x' est un préfixe indiquant que le codage est héxadécimal, puis $80 = 8 \times 16^1 + 0 \times 16^0 = 128$ ($base10$)).
   Une fois le 1^{er} bit lu, on décale le bit 1 du masque d'un cran vers la droite (on suppose qu'on est en train de lire les bits de l'octet `10011101`) :

```
octet        masque       résultat
-------------------------------------
10011101  &  10000000  -> retourne 1
10011101  &  01000000  -> retourne 0
10011101  &  00100000  -> retourne 0
10011101  &  00010000  -> retourne 1
10011101  &  00001000  -> retourne 1
10011101  &  00000100  -> retourne 1
10011101  &  00000010  -> retourne 0
10011101  &  00000001  -> retourne 1
10011101     00000000  -> on lit un nouvel octet
```

Le décalage de bits s'obtient avec les opérateurs suivants :

- `o >> 1` : décalage à droite de 1 cran de l'octet o,
- `o << 1` : décalage à gauche de 1 cran de l'octet o.

4. Testez votre classe en affichant le contenu d'un fichier contenant un caractère ASCII (vous afficherez les bits des 2 octets).

5. Ecrivez une classe `EcrivainOctet` permettant d'écrire un fichier bit par bit (vous attendrez d'avoir 8 bits avant d'écrire dans le fichier). Testez votre classe.

   Indices. Vous pouvez utiliser la classe `FileOutputStream`, dont la méthode `void write(int)` permet d'écrire un octet dans un fichier. L'idée est d'accumuler les bits que l'on souhaite écrire dans le fichier, jusqu'à en avoir 8, puis d'appeler `write`. Vous pouvez utiliser un attribut dont le rôle est de compter le nombre de bits accumulés, et un autre dont le rôle est de stocker les bits (avec décalage à gauche à chaque ajout d'un bit).

   Attention, il est possible qu'on écrive un nombre n de bits tel que n ne soit pas multiple de 8, il faut alors que votre classe contienne une méthode `void vider()` qui ajoute les bits manquants (à 0) pour arriver à 8 bits. Là aussi, nous utiliserons un décalage.
   Enfin, nous noterons l'existence des opérateurs de combinaison d'octets *et* (noté `&`) et *ou* (noté `|`) :

   ```
   00000000 & 00000001 = 00000000
   00000000 | 00000001 = 00000001
   ```