Utilisation des fichiers texte en Java

NFA005

2 mars 2009

1 Introduction

Les fichiers sont des *structures de données stockées sur disque*. A la différence des données gérées en mémoire par les programmes (entiers, booléens, nombres réels, tableaux, objets de différents types, etc), qui sont perdues à la fin du programme, les fichiers sont des structures de données *persistantes*, qu'on peut retrouver la prochaine fois que le programme sera exécutée.

Au niveau physique, un fichier est composé d'une suite de blocs disque de taille constante, géré par le *système d'exploitation*, qui offre aux programmes une vue logique du fichier.

Au niveau logique, le fichier est une suite d'octets, qui représentent des informations de différents types : entiers, booléens, nombres réels, tableaux, objets, etc. Pour pouvoir interpréter ces informations lors de la lecture du fichier, il faut connaître le type d'information qui y a été écrit.

Les fichiers texte sont un cas particulier de fichier, où toute l'information est représentée par une *suite de caractères*. On peut voir un fichier texte comme une longue chaîne de caractères stockée sur disque. Les lignes successives de texte sont séparées par un ou plusieurs caractères de fin de ligne :

- la séquence de deux caractères `'\r'` et `'\n'` en windows.
- le caractère `'\n'` en unix et Mac Intosh.
- le caractère `'\r'` sur les anciens système Apple.

Une différence importante entre les fichiers textes et les fichiers binaires ordinaires est que toute donnée est représentée par *sa chaîne de caractères d'affichage*. Par exemple le boolean `true`, qui normalement est représenté sur un octet, apparaîtra dans le fichier texte en tant que chaîne de caractères *"true"*. Un entier, qui occupe normalement 4 octets, sera représenté par la suite de caractères chiffres qui le composent. Par exemple l'entier `125` sera représenté par la chaîne de caractères *"125"*, tandis que l'entier `-34567` par la chaîne *"-34567"*. Donc des entiers différents, qui en mémoire ont la même taille (4 octets), auront des tailles différentes dans le fichier texte.

Il faut noter que des perturbations liées au codage des caractères peuvent survenir. Il existe en effet deux grands codages :

- le codage ASCII qui est utilisé par les systèmes d'exploitation actuels (unix, windows). Il code chaque caractère avec un octet. Il existe plusieurs variantes permettant de représenter des jeux de caractères différents, notamment pour ce qui concerne les accents.
- le codage unicode, qui est utilisé par Java. Il n'en existe qu'une version unique, mais les polices disponibles sur un ordinateur donné ne couvrent pas tous les caractères unicode. Les caractères sont codés sur deux octets.
- le codage UTF-8 est un codage intermédiaire entre les deux précédents, permettant de stocker de l'unicode dans un système basé sur l'ASCII.

Sur un ordinateur bien configuré, il n'est pas nécessaire de connaître le codage pour faire fonctionner les programmes java. En revanche, il peut y avoir des difficultés à relire sur un ordinateur donné

des fichiers créés sur un autre ordinateur.

Java offre une large palette de classes et méthodes pour utiliser les fichiers, dont nous présenterons ici seulement les plus importantes, à travers des exemples. Toutes ces classes se trouvent dans le paquetage `java.io`, qui doit être importé dans le programme qui utilise les fichiers.

2 Lecture d'un fichier texte

L'exemple suivant demande à l'utilisateur le nom d'un fichier texte, l'ouvre ensuite en lecture, le lit ligne par ligne et l'affiche à l'écran.

```
import java.io.*;

class LectureFichier{
  public static void main(String[] args){
    Terminal.ecrireString("Nom fichier: ");
    String nomFichier = Terminal.lireString();
    try{
      FileReader fr = new FileReader(nomFichier);
      BufferedReader br = new BufferedReader(fr);
      String ligne = br.readLine();
      while(ligne != null){
        Terminal.ecrireStringln(ligne);
        ligne = br.readLine();
      }
      br.close();
    }
    catch(FileNotFoundException e){
      Terminal.ecrireStringln("Fichier non trouvé");
    }
    catch(IOException e){
      Terminal.ecrireStringln("Problème à la lecture du fichier");
    }
  }
}
```

Quelques explications :

- L'instruction `import java.io.*` permet d'utiliser les classes d'accès aux fichiers texte.
- L'objet de type `FileReader` permet de lire un fichier texte caractère par caractère. Si le fichier dont le nom est donnée en paramètre au constructeur n'est pas trouvé, l'exception `FileNotFoundException` est levée.
- L'objet de type `BufferedReader` utilise un tampon mémoire pour éviter d'accéder le disque chaque fois qu'une lecture est demandée. Il optimise les performances et offre des fonctionnalités de lecture plus avancées, comme la lecture ligne par ligne par la méthode `readLine()`.
- Quand l'appel à `readLine()` retourne *null*, la fin du fichier a été atteinte.
- A la fin de la lecture du fichier, il faut fermer le `BufferedReader` avec la méthode `close()`.
- Les méthodes `readLine()` et `close()` peuvent produire une exception `IOException` en cas de problème. Cela n'arrive généralement pas, mais il faut traiter quand même cette exception

dans le programme.

3 Utiliser les chaînes de caractères lues à partir du fichier texte

Il est souvent nécessaire d'utiliser les chaînes de caractères lues à partir du fichier texte pour en extraire des parties, pour les convertir en des données d'autres types, etc.

L'exemple suivant réalise deux traitements de ce genre. Le premier prend une chaîne constituée d'entiers séparés par un ou plusieurs espaces, extrait les sous-chaînes de ces entiers, les convertit en entiers, calcule et affiche leur somme. Le second prend une chaîne où des sous-chaînes sont séparés par un caractère '#', extrait et affiche ces sous-chaînes.

```
class ExtractionString{
  public static void main(String[] args){
    String ex1 = "  23    -17  5 2";
    try{
      int somme = 0;
      String reste = ex1.trim();
      while(reste.length()>0){
        int pos = reste.indexOf(' ');
        String valString;
        if(pos>=0)
          valString = reste.substring(0, pos);
        else valString = reste;
        int val = Integer.parseInt(valString);
        somme = somme + val;
        if(pos>=0)
          reste = reste.substring(pos).trim();
        else reste =  "";
      }
      Terminal.ecrireStringln("Somme = " + somme);
    }
    catch(NumberFormatException e){
      Terminal.ecrireStringln("Entier incorrect");
    }

    String ex2 = " texte# 15 # final";
    String[] parts = ex2.split("#");
    for(int i=0; i<parts.length; i++)
      Terminal.ecrireStringln(parts[i]);
  }
}
```

Quelques explications :

- La méthode `trim()` de `String` retourne une chaîne où les espaces au début et à la fin de la chaîne ont été éliminés.
- La méthode `indexOf` de `String` prend un caractère en paramètre et retourne la première position où l'on trouve ce caractère dans la chaîne (ou -1 si le caractère ne s'y trouve pas).

- La méthode `substring` de `String` extrait une sous-chaîne de la chaîne. Elle prend un ou deux paramètres : le premier est la position de début et le second (s'il existe) est la position tout de suite *après* la sous-chaîne à extraire. Si le second paramètre n'est pas précisé, on extrait jusqu'à la fin de la chaîne.
- La méthode `parseInt` de la classe `Integer`, transforme la chaîne passée en paramètre en une valeur entière. Si la chaîne n'est pas un entier correct, l'exception `NumberFormatException` est levée.
- La technique d'extraction des entiers de la chaîne est la suivante. On élimine les espaces au début et à la fin avec `trim`, ensuite on cherche le premier espace. S'il existe (`pos >= 0`, on extrait la sous-chaîne entier en s'arrêtant à `pos`, sinon c'est toute la chaîne (`reste`) qui représente l'entier. On transforme en entier, on le rajoute à la somme, ensuite on calcule le reste de la chaîne et tout est répété jusqu'à ce que le reste devient vide.
- Pour le second cas, on utilise la méthode `split` de `String`. Elle prend en paramètre une chaîne de caractères qui sert de délimiteur aux sous-chaînes à extraire. Elle retourne un tableau de sous-chaînes, extraites de la chaîne initiale par rapport au délimiteur. Dans notre exemple, le résultat contient 3 sous-chaînes : `"␣texte"`, `"␣15␣"` et `"␣final"`.

4 Lecture d'un fichier texte caractère par caractère

La lecture d'un fichier peut se faire caractère par caractère au lieu de se faire ligne par ligne. Il faut pour cela utiliser la méthode `read` au lieu de la méthode `readLine`. Les opérations d'ouverture et de fermeture du fichier sont identiques.

La méthode `read` ne prend pas de paramètre et renvoie un entier (type `int`). Cet entier est -1 s'il n'y a plus de caractère à lire dans le fichier (la fin de fichier a été atteinte). Sinon il contient le code du caractère dans la table unicode. On peut le convertir en caractère au moyen d'une conversion explicite de type.

Voici le programme d'affichage d'un fichier.

```
import java.io.*;

class LectureFichierBis{
    public static void main(String[] args){
        Terminal.ecrireString("Nom fichier: ");
        String nomFichier = Terminal.lireString();
        try{
            FileReader fr = new FileReader(nomFichier);
            BufferedReader br = new BufferedReader(fr);
            int c = br.read();
            while(c != -1){
                Terminal.ecrireChar((char) c);
                c = br.read();
            }
            br.close();
        }
        catch(FileNotFoundException e){
            Terminal.ecrireStringln("Fichier non trouvé");
        }
```

```
        catch(IOException e){
            Terminal.ecrireStringln("Problème à la lecture du fichier");
        }
    }
}
```

On voit dans ce programme un exemple de conversion d'entier en caractère : `(char) c`. Il est parfois plus agréable de faire une lecture caractère par caractère que ligne par ligne.

5 Ecriture d'un fichier texte

L'exemple suivant demande à l'utilisateur le nom d'un fichier texte, l'ouvre ensuite en écriture et y écrit plusieurs informations.

```
import java.io.*;

class EcritureFichier{
  public static void main(String[] args){
    Terminal.ecrireString("Nom fichier: ");
    String nomFichier = Terminal.lireString();
    try{
      FileWriter fw = new FileWriter(nomFichier);
      BufferedWriter bw = new BufferedWriter(fw);
      PrintWriter pw = new PrintWriter(bw);
      pw.print(125);
      pw.print(" & ");
      pw.println(3.14);
      pw.print("texte");
      pw.close();
    }
    catch(IOException e){
      Terminal.ecrireStringln("Problème à l'écriture du fichier");
    }
  }
}
```

Quelques explications :

- L'objet de type `FileWriter` permet d'écrire un fichier texte caractère par caractère. A la différence de `FileReader`, l'exception `FileNotFoundException` n'est pas levée. Si le fichier existe déjà, il sera écrasé, s'il n'existe pas, il sera créé. Par contre, l'exception `IOException` peut être levée si on ne peut pas créer le fichier (disque plein, pas de droits d'écriture, etc).
- L'objet de type `BufferedWrite` apporte la même optimisation en écriture que `BufferedReader` en utilisant un tampon mémoire. Par contre, les fonctionalités d'écriture restent de bas niveau, d'où le besoin d'un objet `PrintWriter`.
- L'objet `PrintWriter` offre les méthodes `print` et `println` avec un paramètre de n'importe quel type simple (entier, caractère, boolean, double) ou chaîne de caractères. Elles fonctionnent de la même façon que les méthodes `ecrireXXX` et `ecrireXXXln` de la classe `Terminal`.

– A la fin de l'écriture du fichier, il faut fermer le `PrintWriter` avec la méthode `close()`, qui peut produire une exception `IOException`.