Compléments sur les entrées/sorties texte

S. Rosmorduc

juin 2013

Un petit mémo pour fixer les idées

Introduction

On distingue souvent en informatique deux sortes de fichiers : les fichiers *textes* qui contiennent une suite de caractères (et qui sont ceux que nous avons vus en NFA035) et les fichiers binaires qui contiennent une suite d'octets (donc de nombres compris entre 0 et 255).

Un fichier jpeg, un .class, etc... sont autant d'exemples de fichiers binaires. Pour lire un flux binaire en java, on utilise un **InputStream**. Pour écrire un flux binaire, on utilise un **OutputStream**.

Pour faire quelque chose d'intéressant avec un flux binaire, on a généralement besoin de comprendre comment les données y sont codées. Par exemple, on peut imaginer un format binaire d'images où les quatre premiers octets coderaient la taille de l'image, et seraient suivis par les valeurs des pixels (un triplet d'octet par pixel, pour rouge, vert ou bleu).

Bref, dans la plupart des cas, on doit considérer qu'un fichier binaire est un codage numérique d'une donnée. Parmi ces données, il y a le texte. Mais c'est un type tellement fréquent qu'on lui a réservé une place à part. Java permet donc de représenter un **flux texte :** en lecture avec la classe **Reader** ; en écriture avec la classe **Writer**.

Cependant, sur le disque dur, on n'a jamais que des suites d'octets. Un flux texte est donc construit « au dessus » d'un flux binaire.

Pour le programme d'examen, on négligera le problème (pour cette année au moins) ; mais en pratique, il est utile d'avoir quelques connaissances sur la question.

Écriture de flux textes

L'écriture de flux textes est bien plus simple que la lecture... on sait généralement ce qu'on veut écrire.

Pour écrire dans un flux texte, on utilise un **Writer.** Il y a plusieurs classes qui étendent Writer et sont utilisables. Elles ne diffèrent vraiment que par leurs constructeurs.

- FileWriter : permet d'écrire dans un fichier sur le disque ;
- StringWriter : permet d'écrire « en mémoire », et de récupérer la string qui correspond au texte qui a été écrit.

- OutputStreamWriter : permet d'écrire dans un flux binaire. Nous en reparlerons plus tard.

La classe Writer dispose principalement des méthodes suivantes :

- void write(int c) : permet d'écrire un *caractère.* Le type de la variable est « int », mais ça n'a pas une grande importance. On peut écrire :

```
char car= 'b' ;
writer.write(car) ; // écrit 'b'
int car1= 'd' ;
writer.write(car1) ; // écrit 'd'.
```

- void writer(String s) : permet d'écrire une String.
- void close() : ferme le fichier. Il est très important de toujours refermer les fichiers qu'on a ouvert.

L'écriture d'un flux comporte donc trois étapes :

- ouverture (en créant l'objet Writer)
- écriture avec un ou plusieurs appels à write()
- fermeture du flux

Exemple :

```
FileWriter w= new FileWriter("toto.txt") ;
w.write("un petit\ntexte") ;
w.close() ;
```

Va créer le fichier « toto.txt », et y écrire, sur deux lignes (à cause du « \n »), le texte « un petit texte ».

Noter que les méthodes d'entrées/sorties lèvent potentiellement des IOException. Si vous ne savez pas quoi en faire, *laissez-les passer !* (utilisez **throws** dans le doute, de préférence à un try...catch).

La classe FileWriter

La classe FileWriter a deux constructeurs. L'un prend comme argument une String, qui est le chemin du fichier. L'autre constructeur prend comme argument un objet File.

Utilisation de StringWriter :

```
StringWriter w= new StringWriter() ;
w.write("un petit\ntexte") ;
w.close() ;
String s= w.toString();
```

StringWriter est surtout utile quand vous avez une méthode qui attend un Writer et que vous voulez récupérer le résultat en mémoire (par exemple, pour écrire des tests).

Lecture de flux textes

Pour lire un flux texte, on utilise un **Reader**. Il existe plusieurs versions de Reader. En voici quelques-uns

- FileReader : lit dans un fichier
- StringReader : « lit » dans une String (simule un fichier à partir d'une String)
- BufferedReader : ajoute à un reader déjà créé des capacités supplémentaires (nous allons consacrer un petit chapitre à ce type de Readers)

Un algorithme de lecture va typiquement :

- ouvrir le fichier en créant le Reader
- tant qu'on n'est pas arrivé à la fin du fichier, lire les caractères qu'il contient, souvent un par un.
- Fermer le fichier (important!!!)

Il faut imaginer le Reader comme ayant une « tête de lecture », qui va progresser dans le flux. À chaque lecture, cette « tête de lecture » avance d'un cran.

Les deux méthodes de Reader que nous utiliserons ici sont read() et close().

- int read() : fait deux choses à la fois ; renvoie le code du caractère lu, et avance dans le flux. Read() permet aussi de savoir quand on a atteint la fin du fichier. En effet, dans ce cas-là, il renvoie « -1 ». C'est la raison pour laquelle la valeur renvoyée est un « int » et non pas un « char ».

  « -1 » n'est pas une valeur de code possible pour un char java (leur code est positif, compris entre 0 et 65535).
- void close() : ferme le flux.

Typiquement, la lecture d'un fichier complet a la structure suivante :

```
FileReader r= new FileReader("toto.txt") ;
int c= r.read() ; // lecture du premier caractère
// Tant qu'on n'est pas à la fin du fichier...
while (c != -1) {
     char cc= (char)c ;
     System.out.print(cc) ; // affichage du caractère.
     // (Essayez sans passer par un char, pour voir).

     // Lecture du char suivant :
     c= r.read() ;
}
// fermeture.
r.close() ;
```

Attention, « c » doit bien être un « int ». Si, en croyant bien faire, vous utilisez un « char », il ne pourra pas être égal à -1, et votre programme plantera.

Si vous appelez plusieurs fois r.read() dans le corps de votre boucle, vous avancerez dans le fichier à chaque fois. C'est parfois le but recherché... mais ça peut aussi être un bug de votre part.

La classe FileReader

La classe FileReader a deux constructeurs. L'un prend comme argument une String, qui est le chemin du fichier. L'autre constructeur prend comme argument un objet File.

StringReader

Un StringReader est très simple à utiliser (et très pratique pour tester des fonctions qui travaillent sur des flux avec JUnit) :

```
StringReader r= new StringReader("Mon texte ici, bla bla...") ;
int c= r.read() ;
.....
r.close() ;
```

BufferedReader

La classe BufferedReader ajoute à la classe Reader une méthode très intéressante :

- String readLine() : renvoie le contenu de la ligne suivante, ou null en fin de fichier

Cette méthode permet de lire un fichier ligne par ligne.

Le BufferedReader lit ses données dans un autre Reader, qui est passé à son constructeur. Donc, pour lire un fichier ligne par ligne, on va :

- créer un FileReader r0 pour lire dans le fichier
- créer un BufferedReader r qui lira dans r0.

```
FileReader r0= new FileReader("toto.txt") ;
BufferedReader r= new BufferedReader(r0) ;
String ligne= r.readLine() ; // première ligne
// Tant qu'on n'est pas à la fin du fichier...
while (ligne != null) {
     System.out.println(ligne) ; // affichage de la ligne
     // ligne suivante :
     ligne= r.readLine();
}
```

```
// fermeture du BufferedReader (ferme aussi r0).
r.close() ;
```

En pratique, on n'a pas besoin de stocker r0 dans une variable, et on peut remplacer les deux premières lignes par :

```
BufferedReader r= new BufferedReader(new FileReader("toto.txt")) ;
```

Note importante : **le(s) caractères de fin de ligne n'apparaissent pas dans la String renvoyée par readLine().**

Le codage des fichiers textes

Pas de panique pour 2012-2013 : ce qui suit est hors programme. Mais c'est utile en pratique.

Un texte, quand il est écrit dans un flux, l'est avec un certain codage. Sur un ordinateur donné, il y a généralement un codage « par défaut », qui n'est pas toujours le même que celui du voisin. Ainsi, un texte produit sur Mac est souvent codé en MacRoman. Sous certains Unix, il pourra être en ISO 8859 1.

Dans ces deux codages, le « a » a le code 97... mais le « é » a le code 142 en MacRoman, et 233 en ISO 8859 1. Comme 233 en Mac Roman est le code de « È », un fichier ISO 8859 1 lu en croyant qu'il s'agit d'un fichier Mac Roman affichera « È » à la place de « é ».

Ces anciens codages ne comportaient que 256 codes, ce qui était très restrictif. Le codage Unicode commence à les remplacer. Les codes unicodes peuvent utiliser jusqu'à quatre octets; généralement les fichiers unicode sont codés en UTF-8, qui est un *codage à taille variable*.

Bref, c'est compliqué, et il n'y a a priori pas de moyen sûr de trouver le codage d'un fichier texte donné si on ne le connaît pas à l'avance (c'est pour cela que les fichiers HTML 5 ont une balise qui permet de le préciser).

Les sauts de ligne

Pour simplifier les choses, les changements de ligne sont indiqué de manière différente sous Mac OS, Unix, ou Windows. À l'origine, les changements de ligne étaient des codes envoyés aux imprimantes matricielles. Pour changer de ligne, il fallait :

- avancer le papier en tournant le rouleau (« *line feed* », « *new line* ») : code ASCII 10, « \n »
- remettre la tête d'impression en début de ligne (« *carriage return* », ou « retour chariot » en français) : code ASCII 13, « \r ».

Sous Windows, le changement de ligne est conventionnellet indiqué par la séquence « \r\n ». Sur Mac, le « \r » est utilisé, et sous Unix, c'est « \n ».

La plupart des éditeurs de texte comprennent les trois codages du saut de ligne.

Avec un Reader, vous verrez exactement les caractères qui sont dans le fichier. Donc, sur un fichier créé sous Windows, vous observerez successivement un « \r » et un « \n ». Le BufferedReader évite les problèmes.

En pratique

Quand utilisez FileReader et FileWriter, Java utilise le codage par défaut de votre ordinateur.

Le BufferedReader sait lire les trois types de ligne avec readLine() : vous n'avez pas à vous en occuper.

Si vous voulez utiliser un codage donné, par exemple UTF-8, vous pouvez :

- créer un fichier binaire avec FileInputStream (lecture) ou FileOutputStream (écriture)
- construire un flux texte par dessus ce flux binaire avec InputStreamReader (lecture) ou OutputStreamWriter (écriture). Ces deux classes ont un constructeur qui permet de spécifier le codage (si quelqu'un nous expliquer pourquoi FileReader et FileWriter n'ont pas de tel constructeur, nous sommes intéressés, merci).

Exemple : écriture d'un fichier en UTF-8 :

```
FileOutputStream o= new FileOutputStream("toto.txt") ;
OutputStreamWriter w= new OutputStreamWriter(o, "UTF-8") ;
w.write("Bien élevé. αβγ.");
w.close() ;
```