Entrées/sorties

Michel Schinz

2018-03-05

1 Introduction

En informatique, le terme **entrées/sorties** (*input/output*) — abrégé E/S ou I/O en anglais — désigne les échanges de données qui ont lieu entre un programme et le monde extérieur. Dans la plupart des cas, les données sont échangées entre le programme et :

- un ou plusieurs fichiers, et/ou
- l'utilisateur — via la console ou terminal — et/ou
- une ou plusieurs connexions réseau.

Cette leçon se concentre sur les entrées/sorties via des fichiers mais la plupart des notions s'appliquent également aux entrées/sorties via la console ou le réseau.

2 Fichier et système de fichiers

Un **fichier** (*file*) est une séquence d'octets (*bytes*) stockée sur un disque. De plus en plus, ce « disque » est en fait une forme de mémoire permanente, p.ex. une mémoire flash.

Le contenu d'un fichier est accompagné de **méta-données** (*metadata*) donnant différentes informations sur le fichier : son nom, ses droits d'accès, la date et l'heure de la dernière modification, etc.

Un **système de fichiers** (*file system*) permet d'organiser un ensemble de fichiers dans des **répertoires** (*directories*) qui peuvent être imbriqués les uns dans les autres, donnant lieu à une structure arborescente. Dans un tel système, un fichier ou répertoire est désigné par son **chemin d'accès** (*access path* ou simplement *path*), composé de la liste des répertoires dans lesquels il est imbriqué, et de son nom.

Dans la plupart des systèmes d'exploitation, les chemins d'accès sont représentés par des chaînes de caractères dans lesquelles les noms des différentes composantes du chemin (répertoires et fichier) apparaissent dans l'ordre d'imbrication, séparées par un caractère de séparation.

Par exemple, les systèmes Unix (Linux, macOS, etc.) utilisent la barre oblique (`/`, *slash* en anglais) comme caractère de séparation. Ainsi, le chemin `/usr/bin/awk` désigne le fichier (ou répertoire) nommé `awk`, imbriqué dans un répertoire nommé `bin`, lui-même imbriqué dans `usr`. La barre oblique initiale désigne la racine de la hiérarchie.

Windows utilise quant à lui la barre oblique inversée (`\`, *backslash* en anglais) comme caractère de séparation, et inclut en tête un « nom de lecteur » (*drive name*) identifiant le disque auquel appartient le chemin. Ainsi, le chemin Windows `C :\Program Files\Paint` désigne le fichier (ou répertoire) nommé `Paint`, imbriqué dans un répertoire nommé `Program Files` se trouvant sur le lecteur (généralement un disque) `C`.

2.1 Types de fichiers

On fait généralement la distinction entre deux catégories de fichiers :

1. les fichiers dits **textuels** (*text files*), qui contiennent une séquence de caractères encodés, et
2. les fichiers dits **binaires** (*binary files*), qui sont tous les autres fichiers : images, sons, formats propriétaires utilisés par différents programmes, etc.

Bien que répandue, cette terminologie n'est pas très cohérente dans la mesure où tout fichier, même s'il contient du texte, est composé d'une suite d'octets et est donc « binaire ».

3 Entrées/sorties en Java

La bibliothèque Java possède de nombreuses classes dédiées aux entrées/sorties qui, pour des raisons historiques, sont réparties entre deux paquetages :

- `java.io` contient les classes d'entrées/sorties d'origine,
- `java.nio` (pour *new io*) contient des classes plus récentes.

Malgré ce que ces noms pourraient suggérer, les classes de `java.nio` ne remplacent pas celles de `java.io`. Les deux sont nécessaires dans différentes situations, et sont même souvent utilisées simultanément.

Le paquetage `java.nio` a plusieurs buts, son principal étant d'offrir une abstraction de base différente — et plus efficace — que celle de `java.io`. Comme nous le verrons, l'abstraction de base de `java.io` est le **flot** (*stream*), tandis que celle de `java.nio` est la **mémoire tampon** (*buffer*). En général, `java.io` est plus simple à comprendre et à utiliser, et ses performances assez bonnes pour la plupart des cas. Nous n'étudierons donc pas `java.nio` en détail ici.

3.1 Flot d'entrée/sortie

L'abstraction de base des entrées/sorties du paquetage `java.io` est le flot. Un **flot** (*stream*) est une séquence de valeurs — qui peuvent p.ex. être des octets ou des caractères — auxquelles on accède de manière séquentielle, c-à-d de la première à la dernière.

Un **flot d'entrée** (*input stream*) est un flot dont les valeurs proviennent d'une source donnée — fichier, connection réseau, etc. — et sont lues par le programme.

Un **flot de sortie** (*output stream*) est un flot dont les valeurs sont fournies par le programme et écrites dans un « puit » donné — fichier, connection réseau, etc — qui les absorbe.

La bibliothèque Java distingue deux types de flots :

1. les flots d'octets,
2. les flots de caractères.

La nomenclature peut toutefois prêter à confusion puisque les flots d'octets sont nommés *streams* — *input streams* pour ceux d'entrée, *output streams* pour ceux de sortie — tandis que les flots de caractères sont nommés *readers* (lecteurs) pour ceux d'entrée et *writers* (écrivains) pour ceux de sortie.

A chacun de ces quatre types de flots correspond une hiérarchie de classes séparée dans la bibliothèque Java. La classe au sommet de la hiérarchie dans chacun de ces quatre cas est donnée par la table suivante :

	Entrée	Sortie
Octets	`InputStream`	`OutputStream`
Caractères	`Reader`	`Writer`

3.2 Flots d'entrée

La classe abstraite `InputStream` du paquetage `java.io` représente un flot d'entrée d'octets.

```
abstract public class InputStream {
  int read();
  int read(byte[] b);
  int read(byte[] b, int o, int l);

  long skip(long n);
  int available();

  boolean markSupported();
  void mark(int l);
  void reset();
```

```
  void close();
}
```

Elle possède plusieurs sous-classes représentant soit :

- des flots d'entrée primaires, qui obtiennent directement leurs données d'une source comme un fichier, une connexion réseau, etc.
- des flot d'entrée dits filtrants, qui obtiennent leurs données d'un autre flot et les transforment à la volée.

3.2.1 Méthodes de lecture

Les trois variantes de la méthode `read` permettent de lire — et, ce faisant, de consommer — un ou plusieurs octets :

- `int read()` : lit et retourne le prochain octet sous la forme d'une valeur comprise entre 0 et 255 inclus, ou –1 si la fin du flot a été atteinte,
- `int read(byte[] b, int o, int l)` : lit au plus `l` octets du flot, les place dans le tableau `b` à partir de la position `o` et retourne le nombre d'octets lus,
- `int read(byte[] b)` équivaut à `read(b, 0, b.length)`.

La méthode `skip` permet d'ignorer des octets :

- `long skip(long n)` : ignore au plus `n` octets du flot et retourne le nombre d'octets ignorés.

Les méthodes `read` et `skip` sont potentiellement **bloquantes** (*blocking*). Cela signifie que si la totalité des octets demandé à ces méthodes n'est pas encore disponible mais le sera dans le futur — p.ex. après avoir été lus du disque ou reçus du réseau — le programme est bloqué jusqu'à l'arrivée des octets.

Cette caractéristique facilite la programmation mais peut nuire aux performances ou à l'interactivité de l'application. L'un des buts de `java.nio` est d'ailleurs de fournir des méthodes permettant l'accès non bloquant aux données.

Pour éviter de bloquer le programme, il peut être utile de connaître le nombre d'octets qu'il est possible d'obtenir du flot sans blocage. C'est le but de la méthode `available` :

- `int available()` : retourne une estimation du nombre d'octets qu'il est possible de lire ou d'ignorer sans bloquer.

Il est garanti qu'un appel à `read` ou `skip` demandant le nombre d'octets retourné par `available` ne bloquera pas, mais il n'est par contre pas garanti que ce nombre d'octets sera effectivement lu ou ignoré.

3.2.2 Représentation des octets

Pour mémoire, le type `byte` de Java représente un octet signé dont la valeur est comprise entre -128 et +127. Cela implique une différence importante entre les deux principales variantes de la méthode `read` :

- celle qui retourne le prochain octet en tant qu'entier `int` le retourne sous forme d'une valeur comprise entre 0 et 255 inclus, -1 étant utilisé pour signaler la fin du flot,
- celle qui stocke les octets dans un tableau de type `byte[]` les stocke sous forme de valeurs comprises entre -128 et +127 inclus, -1 étant une valeur normale n'indiquant aucunement la fin du flot !

Par exemple, étant donné un flot `s` contenant les deux octets suivants (donnés ici en binaire) :

```
00000000 11111111
```

Le morceau de programme ci-dessous :

```
System.out.println(s.read() + ”,” + s.read());
```

affiche `0,255` tandis que celui ci-dessous

```
byte[] b = new byte[2];
s.read(b);
System.out.println(b[0] + ”,” + b[1]);
```

affiche `0,-1`.

3.2.3 Méthodes de marquage

Les méthodes de marquage, peu utilisées, permettent de marquer une position dans le flot et d'y retourner plus tard :

- `boolean markSupported()` : retourne vrai si et seulement si le flot gère le marquage,
- `void mark(int l)` : place la marque à l'endroit actuel et promet que si la méthode `reset` est appelée plus tard, elle le sera lorsqu'au plus `l` octets auront été lus depuis la position actuelle,
- `void reset()` : revient à la position marquée précédemment.

3.2.4 Méthode de fermeture

Souvent, à un flot est associé une resource du système d'exploitation sous-jacent. Par exemple, l'ouverture d'un fichier ou la création d'une connexion réseau provoquent l'allocation de mémoire au niveau du système, mémoire qui n'est libérée que lorsque le fichier ou la connexion sont fermés.

Pour cette raison, les flots ont tous une méthode `close` permettant de fermer le flot et de libérer les éventuelles resources du système qui lui sont associées :

- `void close()` : ferme le flot, libérant ainsi les éventuelles resources associées et rendant par là même le flot inutilisable.

Attention : après fermeture, un flot est inutilisable et l'appel de n'importe laquelle de ses méthodes — autre que `close` — provoque une exception.

3.3 Flots d'entrée concrets

Les sous-classes concrètes de `InputStream` sont réparties en deux familles :

1. celles qui représentent des flots d'entrée primaires, dont les données proviennent d'une source déterminée (fichier ou autre) et qui héritent généralement directement de `InputStream`,
2. celles qui représentent des flots d'entrée filtrants, dont les données proviennent d'un flot sous-jacent et qui héritent généralement de `FilterInputStream`, elle-même sous-classe de `InputStream`.

3.3.1 Flot d'entrée de fichier

La classe `FileInputStream` représente un flot d'entrée primaire dont les octets proviennent d'un fichier.

Par exemple, un flot fournissant les octets du fichier nommé `file.bin` peut s'obtenir ainsi :

```
InputStream s = new FileInputStream(”file.bin”);
```

3.3.2 Flot d'entrée de tableau

La classe `ByteArrayInputStream` représente un flot d'entrée primaire dont les octets proviennent d'un tableau Java de type `byte[]`.

Par exemple, un flot fournissant les octets 1, 2 puis 3 peut s'obtenir ainsi :

```
byte[] b = new byte[]{ 1, 2, 3 };
InputStream s = new ByteArrayInputStream(b);
```

Un autre constructeur existe pour ne fournir que les octets dont les index sont dans un intervalle donné. (`ByteArrayInputStream` peut être assez utile pour écrire des tests unitaires.)

3.3.3 Flot à mémoire tampon

La classe `BufferedInputStream` représente un flot filtrant — qui ne filtre rien du tout, en réalité — ajoutant une mémoire tampon au flot sous-jacent. Cette classe lit les octets du flot sous-jacent par blocs, les stocke dans un tableau, puis les fournit lorsqu'on les lui demande.

L'utilisation d'une telle mémoire tampon au-dessus d'un flot dont les données sont coûteuses à obtenir une à une — p.ex. car elles proviennent d'un disque lent — peut améliorer les performances. Elle est donc souvent utilisée sur un flot de type `FileInputStream`, ainsi :

```
InputStream s =
  new BufferedInputStream(new FileInputStream…());
```

De plus, la classe `BufferedInputStream` gère le marquage, donc ses méthodes `mark` et `reset` peuvent donc être utilisées pour marquer une position lors de la lecture et y revenir ultérieurement. Bien entendu, sa méthode `markSupported` retourne vrai.

3.3.4 Flot décompressant

La classe `GZIPInputStream` — du paquetage `java.util.zip` — décompresse à la volée les données fournies par le flot sous-jacent au moyen de l'algorithme gzip.

En d'autres termes, son flot sous-jacent doit fournir des données compressées au format gzip, qu'elle fournit à son client — via la méthode `read` — sous forme décompressée.

3.3.5 Empilement de flots

Il est très fréquent d'empiler plusieurs flots filtrants sur un flot primaire. Par exemple, pour lire un fichier nommé `in.gz` compressé avec gzip, en utilisant une mémoire tampon, on peut écrire :

```
InputStream s =
  new GZIPInputStream(
    new BufferedInputStream(
      new FileInputStream(”in.gz”)));
```

Dans un tel empilement, la fermeture du flot au sommet — ici référencé par `s` — provoque la fermeture de la totalité des flots qui le composent.

3.3.6 Exemple

L'extrait de programme ci-dessous compte le nombre d'octets nuls dans le fichier `in.gz`, compressé avec gzip :

```
InputStream s =
  new GZIPInputStream(
    new BufferedInputStream(
      new FileInputStream("in.gz")));
int b, c = 0;
while ((b = s.read()) != -1) {
  if (b == 0)
    c += 1;
}
s.close();
System.out.println(c);
```

3.4 Flots de sortie

La classe abstraite `OutputStream` du paquetage `java.io` représente un flot de sortie d'octets.

```
abstract public class OutputStream {
  void write(int b);
  void write(byte[] b);
  void write(byte[] b, int o, int l);

  void flush();

  void close();
}
```

Tout comme `InputStream`, elle possède deux grandes catégories de sous-classes, celles représentant des flots de sortie primaires — à destination de fichiers p.ex. — et celles représentant des flots de sortie filtrants, qui envoient les données à un flot sous-jacent en les modifiant éventuellement au passage.

3.4.1 Méthodes d'écriture

Les trois variantes de la méthode `write` permettent d'écrire un ou plusieurs octets dans le flot :

- `void write(int b)` : écrit les 8 bits de poids faible de `b` dans le flot (les 24 bits de poids fort sont ignorés),

- `void write(byte[] b, int o, int l)` : écrit les `l` octets obtenus du tableau `b` à partir de la position `o` dans le flot,
- `void write(byte[] b)` équivaut à `write(b, 0, b.length)`.

Les variantes de la méthode `write` ne garantissent pas que les données sont écrites immédiatement dans le flot. Pour s'en assurer, on peut utiliser la méthode `flush` :

- `void flush()` : force les données du flot à être écrites effectivement, p.ex. sur le disque ou sur la console.

3.4.2 Méthode de fermeture

Tout comme les flots d'entrée, les flots de sortie possèdent une méthode `close` permettant de fermer le flot :

- `void close()` : ferme le flot, libérant ainsi les éventuelles resources associées et rendant par là même le flot inutilisable.

3.4.3 Flots de sortie concrets

La hiérarchie des sous-classes de `OutputStream` est très similaire à celle des sous-classes de `InputStream`.

Toutes les classes de flots d'entrée que nous avons décrites ont un équivalent dans le monde des flots de sortie. Par exemple, à la classe `FileInputStream` correspond la classe `FileOutputStream`, et ainsi de suite. Ces classes correspondant directement à d'autres déjà examinées ne le seront plus ici.

La classe `PrintStream` ajoute à la classe `OutputStream` des méthodes `print`, `println` et `printf` permettant d'écrire des représentation textuelles de données Java dans le flot. Les flots représentant la « sortie standard » (*standard output*) et « l'erreur standard » (*standard error*), stockés respectivement dans les attributs statiques `out` et `err` de la classe `System` sont de type `PrintStream`.

3.5 Resources

On l'a vu, il est important que les flots — lecteurs et écrivains compris — soient fermés au moyen de la méthode `close` lorsqu'ils sont devenus inutiles. De manière générale, on appelle **resources** de tels objets liés à une resource du système d'exploitation et qui doivent être fermés en fin d'utilisation. L'utilisation de telles resources est délicate car il est facile d'oublier l'appel à leur méthode `close` ou de mal gérer le cas où une exception est levée.

Par exemple, le morceau de code ci-dessous utilise une resource et appelle bien la méthode `close`. Est-il pour autant correct ?

```
InputStream s = new FileInputStream(”…”);
System.out.println(s.available());
s.close();
```

Non, ce code n’est pas totalement correct car si la méthode `available` lève une exception — ce qui est possible — la méthode `close` ne sera pas appelée… Ce problème peut être corrigé au moyen d’un bloc `try … finally`, mais le code s’en trouve passablement alourdi :

```
InputStream s = new FileInputStream(”…”);
try {
  System.out.println(s.available());
} finally {
  s.close();
}
```

Java offre depuis peu une variante de l’énoncé `try` nommé *try-with-resource* qui permet d’écrire très simplement du code utilisant une resource. En l’utilisant, notre exemple devient simplement :

```
try (InputStream s = new FileInputStream…()) {
  System.out.println(s.available());
}
```

La méthode `close` ne doit plus être appelée explicitement, cela est fait automatiquement à la fin du bloc `try`.

Un énoncé *try-with-resource* peut être utilisé avec plusieurs variables resources, et suivi d’éventuels clauses `catch` et `finally`, comme la version simple de l’énoncé `try`. P.ex. :

```
try (InputStream i = new FileInputStream(”in.bin”);
     OutputStream o = new FileOutputStream(”out.bin”)) {
  // … code utilisant les flots i et o
} catch (IOException e) {
  // … code gérant l’exception
} finally {
  System.out.println(”done!”);
}
```

Pour pouvoir être utilisée dans un énoncé *try-with-resource*, une valeur doit implémenter l’interface `AutoCloseable` du paquetage `java.lang`, définie ainsi :

```
public interface AutoCloseable {
  void close();
}
```

Cette interface est implémentée par toutes les classes de l'API Java possédant une méthode `close`, parmi lesquelles figurent `InputStream`, `OutputStream`, `Reader` et `Writer`.

4 Représentation des caractères

Comme n'importe quelle donnée, les caractères doivent être représentés sous forme binaire pour pouvoir être manipulés par un ordinateur.

En théorie, comme il n'existe qu'un nombre fini de caractères, leur représentation est simple : il suffit d'en établir une liste exhaustive et de représenter chaque caractère par son index dans cette liste, appelé son code.

En pratique, au vu du grand nombre de caractères existant dans le monde — et de l'invention de nouveaux caractères comme les émoticônes — la simple définition de cette liste est une tâche considérable, actuellement encore en cours.

4.1 Représentations régionales

Historiquement, de nombreuses représentations des caractères ont été établies pour des sous-ensembles des caractères existants, typiquement ceux utilisés par un petit groupe de langues similaires. Par exemple, pour l'anglais, seule une centaine de caractères sont nécessaires. Les premières normes de représentation des caractères — comme la célèbre norme ASCII — ne pouvaient représenter que ces caractères-là.

La norme **ASCII** (*American Standard Code for Information Interchange*) représente un caractère par un entier de 7 bits, et permet donc d'en représenter $2^7 = 128$ différents. Cette norme américaine a été conçue pour représenter les textes anglais et inclut donc les lettres non accentuées de l'alphabet latin, les chiffres et la ponctuation. De plus, certaines valeurs permettent de représenter des pseudo-caractères dits **caractères de contrôle** (*control characters*). Par exemple, l'entier 9 encode un saut de tabulation, l'entier 10 un saut de ligne (*line feed*), etc.

La norme ASCII ne permet pas de représenter tous les caractères de toutes les langues utilisant l'alphabet latin. Par exemple, elle n'inclut pas les lettres accentuées, les ligatures (p.ex. œ), le symbole monétaire de l'Euro (), etc. De nombreuses extensions d'ASCII à 8 bits ont donc été inventées, utilisant la plage des valeurs de 128 à 255 pour ces caractères manquants. Ces extensions sont généralement conçues pour un petit ensemble de langues similaires.

Parmi les extensions 8 bits d'ASCII assez bien adaptées au français et encore en usage dans les pays francophones, on peut citer :

- **ISO 8859–1**, ou **ISO latin 1**, malheureusement incapable de représenter les caractères , œ ou Œ,
- **ISO 8859–15**, variante de 8859–1 résolvant ce problème,

- **Mac Roman**, utilisé sur Mac OS (OS X),
- **Windows 1252**, variante de ISO 8859–1 utilisée sur Windows.

Ces encodages régionaux et/ou spécifiques à un système d'exploitation sont en train de disparaître au profit d'Unicode.

4.2 Unicode

Le standard **Unicode** a pour but d'être universel, c-à-d de permettre la représentation de la totalité des « caractères » existants, y compris les symboles graphiques et mathématiques, les émoticônes, etc. Un caractère Unicode est représenté par un entier appelé son **point de code** (*code point*) et compris :

1. soit entre 0_{16} et $D7FF_{16}$,
2. soit entre $E000_{16}$ et $10FFFF_{16}$.

Ces deux plages contiennent un peu plus d'un million de valeurs différentes. Le trou entre $D800_{16}$ et $DFFF_{16}$ est réservé pour l'encodage UTF–16 décrit plus loin.

Contrairement aux standards 7/8 bits antérieurs précités, dans lesquels chaque caractère est représenté par l'octet de son code, Unicode propose trois encodages différents pour ses caractères :

- les deux premiers, **UTF–8** et **UTF–16**, sont à longueur variable, c-à-d qu'un caractère est représenté par un nombre d'octets qui dépend de son code,
- le dernier, **UTF–32**, est à taille fixe, c-à-d qu'un caractère est toujours représenté par un nombre fixe d'octets, 4 dans ce cas — donc 32 bits.

UTF signifie *Unicode Transformation Format*.

4.2.1 UTF-8

UTF–8, encodage d'Unicode à taille variable, est le plus fréquemment rencontré en pratique. Un caractère y est représenté par une séquence de 1 à 4 octets, selon la plage à laquelle il appartient :

- 1 octet : de 0_{16} à $7F_{16}$,
- 2 octets : de 80_{16} à $7FF_{16}$,
- 3 octets : de 800_{16} à $FFFF_{16}$,
- 4 octets : de 10000_{16} à $10FFFF_{16}$.

Etant donné que les 128 premiers caractères d'Unicode sont ceux de ASCII (dans le même ordre), une chaîne composée uniquement de caractères représentables en ASCII a le même encodage en ASCII qu'en UTF–8.

La chaîne « œuf à 1 » comporte 8 caractères dont les points de codes Unicode hexadécimaux (base 16) sont :

œ	153
u	75
f	66
	20
à	E0
	20
	20AC
1	31

Le nombre d'octets nécessaire à l'encodage de chacun de ces points de code en UTF-8 varie de 1 à 3 :

œ	C5 93
u	75
f	66
	20
à	C3 A0
	20
	E2 82 AC
1	31

Comme on le voit, les caractères qui existent aussi dans la norme ASCII — ici u, f, 1 et l'espace — sont encodés exactement comme en ASCII, sur un octet.

4.2.2 UTF-16

UTF–16 est un autre encodage à taille variable d'Unicode. Un caractère y est représenté par une séquence de 2 ou 4 octets, selon la plage à laquelle il appartient :

- 2 octets : de 0_{16} à $D7FF_{16}$ et de $E000_{16}$ à $FFFF_{16}$
- 4 octets : de 10000_{16} à $10FFFF_{16}$

La première plage couvre la quasi-totalité des alphabets en usage actuellement, donc dans la plupart des cas un caractère Unicode est encodé par un unique mot de 16 bits — donc 2 octets — en UTF–8. Les émoticônes, qui appartiennent à la seconde plage, sont une exception notable.

Tous les caractères de la chaîne « œuf à 1 » sont dans la première plage UTF-16, donc représentables chacun par deux octets :

œ	01 53
u	00 75
f	00 66
	00 20
à	00 E0
	00 20
	20 AC
1	00 31

4.2.3 UTF-32

UTF–32 est un encodage à taille fixe d'Unicode. Il consiste à trivialement représenter chaque caractère au moyen de son point de code, sur 32 bits — donc 4 octets. La chaîne « œuf à 1 » s'encode donc en UTF–32 au moyen de 32 octets :

œ	00 00 01 53
u	00 00 00 75
f	00 00 00 66
	00 00 00 20
à	00 00 00 E0
	00 00 00 20
	00 00 20 AC
1	00 00 00 31

4.2.4 Comparaison UTF-8, 16 et 32

Les chaînes écrites dans des langues utilisant l'alphabet latin sont représentées de manière plus compacte avec l'encodage UTF–8 qu'avec UTF–16 ou UTF–32. Cela est visible avec la chaîne d'exemple « œuf à 1 » qui s'encode au moyen de :

- 12 octets en UTF–8,
- 16 octets en UTF–16 et
- 32 octets en UTF–32.

Cette économie et la compatibilité avec ASCII — qui permet à d'anciens outils comme `grep` sur Unix de fonctionner plus ou moins sur les fichiers encodés en UTF–8 — font d'UTF–8 l'encodage le plus populaire pour Unicode.

5 Représentation des caractères en Java

Lorsque Java a été conçu, le standard Unicode représentait les caractères par 16 bits uniquement. Pour cette raison, le type `char` en Java est défini comme comportant 16 bits.

Dès lors, un caractère Unicode dont le point de code est supérieur à $FFFF_{16}$ ne peut pas être représenté dans un caractère Java de type `char` ! Il en va par exemple ainsi de toutes les émoticônes, comme 👍 dont le code est $1F44D_{16}$.

Les chaînes Java — c-à-d la classe `String` — sont définies pour des raisons historiques comme étant des séquences de « caractères » de type `char`. Ces « caractères » sont en fait les valeurs des entiers de 16 bits constituant l'encodage en UTF–16 de la chaîne.

Toute chaîne composée uniquement de caractères de la première plage UTF–16 (de 0 à $FFFF_{16}$) sera donc composée d'autant de « caractères » de type `char` qu'il y a de caractères Unicode dans la chaîne. Cette correspondance n'existe plus dès qu'une chaîne comporte des caractères dont le code est hors de cette plage, comme les émoticônes.

Tous les caractères Unicode peuvent être inclus tels quels dans une chaîne littérale Java, p.ex.

```
String s = ”œuf␣à␣€1”;
```

Ces caractères peuvent également être écrits au moyen de la syntaxe d'**échappement Unicode** (*Unicode escapes*) en utilisant leur code Unicode à 4 chiffres hexadécimaux, précédés de `\u`. Par exemple, le code Unicode de € étant $20AC_{16}$, la chaîne ci-dessus peut aussi s'écrire :

```
String s = ”œuf␣à␣\u20AC1”;
```

Les caractères faisant partie de la seconde plage UTF–16 (au-delà de $FFFF_{16}$) ne peuvent être placés dans une chaîne Java littérale que tels quels ou au moyen d'une séquence de deux échappements Unicode. Par exemple, le point de code de l'émoticône 👍 ($1F44D_{16}$) s'encode en UTF–16 par les deux entiers de 16 bits successifs $D83D_{16}$ $DC4D_{16}$. On peut donc afficher 👍 au moyen du programme suivant :

```
String thumbsUp = ”\uD83D\uDC4D”;
System.out.println(thumbsUp);
```

5.1 Méthodes des chaînes

La plupart des méthodes de la classe `String` travaillent en termes de « caractères » Java — c-à-d en termes d'entiers de 16 bits de l'encodage UTF–16 — et pas en termes de caractères Unicode. Ainsi, les index passés à `charAt`, `substring` et autres sont toujours exprimés en termes de ces « caractères » Java. Des méthodes ont été ajoutées récemment pour travailler en termes de caractères Unicode.

Par exemple, la méthode `length` retourne le nombre de caractères Java de la chaîne, qui est plus grand ou égal au nombre de caractères Unicode qu'elle possède, que l'on peut obtenir avec la méthode `codePointCount`.

La différence entre le comportement des différentes méthodes de String peut s'illustrer avec la chaîne composée de l'émoticône 👍 :

```
String thumbsUp = ”\uD83D\uDC4D”;
```

Cette chaîne contient un seul caractère Unicode, mais deux « caractères » Java :

```
thumbsUp.length();              // 2
thumbsUp.charAt(0);             // 0xD83D
thumbsUp.charAt(1);             // 0xDC4D
thumbsUp.codePointCount(0, 2); // 1
thumbsUp.codePointAt(0);        // 0x1F44D
```

Le paquetage `java.nio.charset` — que nous n'examinerons pas en détail — contient un ensemble de classes permettant de travailler avec des encodages de caractères. Ainsi :

- la classe `Charset` représente un encodage de caractères, et celui utilisé par défaut peut être obtenu au moyen de la méthode statique `defaultCharset`, et
- la classe `StandardCharsets` de ce paquetage donne accès, via des attributs statiques, aux encodages les plus utiles en pratique — ASCII, ISO 8859–1, UTF–8 et trois variantes de UTF–16.

6 Entrées/sorties textuelles

On appelle **fichier textuel** (*text file*) un fichier ne contenant qu'une séquence de caractères encodés en octets au moyen d'un encodage donné — ASCII, UTF–8 ou autre.

A noter que l'identité de l'encodage utilisé n'est généralement pas attachée au fichier et doit donc être connue par un moyen externe, p.ex. le type MIME pour les pages Web. En l'absence d'une telle information, il faut utiliser une valeur par défaut ou tenter de deviner l'encodage, mais la probabilité de se tromper est grande car, dans le cas général, il n'existe pas de méthode permettant de déterminer l'encodage d'un fichier textuel en examinant son contenu. Par exemple, un fichier contenant la séquence d'octets :

```
62 c5 93 75 66
```

peut être correctement décodé en :

- UTF–8, pour obtenir le texte « bœuf »,
- ISO 8859–1, pour obtenir le texte « bÅ uf »,
- Mac Roman, pour obtenir le texte « bìuf »,
- Windows 1252, pour obtenir le texte « bÅ“uf »,
- etc.

Même si dans le cas général il n'est pas possible d'identifier avec certitude l'encodage d'un fichier textuel, on note que :

1. certaines séquences d'octets sont invalides dans certains encodages à longueur variable comme UTF–8 ou UTF–16, permettant — si on les rencontre — d'éliminer ces candidats,

2. au moyen de statistiques, on peut déterminer qu'un décodage produisant la chaîne « bœuf » a plus de chances d'être correct qu'un décodage produisant la chaîne « bÅ“uf ».

En dehors de l'encodage, un autre aspect variable des fichiers textuels est la représentation des fins de lignes. Les trois représentations que l'on trouve aujourd'hui sont :

1. le retour de chariot (*carriage return* ou CR), dont le code est 13 en ASCII, UTF–8 et autres,

2. le saut de ligne (*line feed* ou LF), dont le code est 10 en ASCII, UTF–8 et autres,

3. le retour de chariot suivi du saut de ligne (souvent abrégée CRLF).

La première représentation était utilisée sur Mac OS avant la version 10 et est donc devenu rare, la seconde est utilisée sur les systèmes Unix et leurs dérivés (Linux, OS X), la dernière est utilisée par Windows.

6.1 Entrées/sorties textuelles en Java

Dans la bibliothèque Java, un **lecteur** (*reader*) est un flot d'entrée de caractères ; un **écrivain** (*writer*) est un flot de sortie de caractères.

Dans le monde des lecteurs/écrivains, la classe `Reader` joue le rôle de `InputStream`, tandis que la classe `Writer` joue celui de `OutputStream`. Ces deux (paires de) hiérarchies de classes sont très similaires, et les concepts se retrouvent, avec des noms similaires. Par exemple, `FileReader` est l'équivalent de `FileInputStream`.

Les lecteurs qui doivent détecter les fins de ligne considèrent généralement que n'importe laquelle des trois séquences suivantes termine une ligne :

- CR (retour de chariot),
- LF (saut de ligne),
- CR puis LF.

Les écrivains qui doivent écrire une fin de ligne utilisent la propriété système nommée `line.separator`, une chaîne dont la valeur peut s'obtenir par l'appel suivant :

```
System.getProperty(”line.separator”);
```

6.1.1 Lecteurs

La classe `Reader` est l'équivalent de `InputStream` mais travaille sur des caractères Java plutôt que des octets :

```
abstract public class Reader {
  int read();
  int read(char[] c, int o, int l);
  int read(char[] c);

  long skip(long n);
  boolean ready();

  boolean markSupported();
  void mark(l);
  void reset();

  void close();
}
```

Contrairement à `InputStream`, `Reader` ne possède pas de méthode `available` permettant de connaître une estimation du nombre de caractères pouvant être lus sans bloquer. Au lieu de cela, elle possède la méthode `ready` qui retourne vrai s'il est certain que le prochain appel à `read` sans argument ne bloquera pas. La raison de cette différence est que fournir une méthode `available` demanderait de décoder les octets disponibles en caractères, ce qui peut être coûteux.

6.1.2 Ecrivains

La classe `Writer` est l'équivalent de `OutputStream` mais travaille sur des caractères Java et pas des octets :

```
abstract public class Writer {
  void write(int c);
  void write(char[] a);
  void write(char[] a, int o, int l);
  void write(String s);

  Writer append(char c);
  Writer append(CharSequence c);
  Writer append(CharSequence c, int s, int e);

  void flush();

  void close();
```

```
}
```

6.1.3 Flots et lecteurs/écrivains

Le lien entre les lecteurs/écrivains et les flots d'octets est fait par deux classes :

- `InputStreamReader` qui, étant donnés un flot d'entrée d'octets et un encodage de caractères, fournit un lecteur, et
- `OutputStreamWriter` qui, étant donnés un flot de sortie d'octets et un encodage de caractères, fournit un écrivain.

6.1.4 Lecteurs/écrivains de fichiers

La classe `FileReader` représente un lecteur dont les caractères proviennent d'un fichier. Cette classe est malheureusement limitée dans la mesure où elle utilise toujours l'encodage par défaut. Pour en utiliser un autre, il convient de lire le fichier au moyen d'un `FileInputStream` puis de convertir ce dernier en lecteur au moyen d'un `InputStreamReader`.

La classe `FileWriter` représente un écrivain dont les caractères sont écrits dans un fichier. Elle souffre de la même limitation concernant l'encodage que `FileReader`.

6.1.5 Lecteurs/écrivains de chaînes

La classe `CharArrayReader` représente un lecteur dont les caractères proviennent d'un tableau de caractères Java de type `char[]` passé au constructeur.

La classe `CharArrayWriter` représente un écrivain écrivant ses caractères dans un tableau de caractères Java de type `char[]`. Celui-ci peut être obtenu au moyen de la méthode `toCharArray`.

Les classes `StringReader` et `StringWriter` sont similaires mais basées sur des chaînes de caractères Java de type `String` plutôt que sur des tableaux.

6.1.6 Lecteurs/écrivains à mémoire tampon

A l'image de `Buffered…Stream`, les classes `BufferedReader` et `BufferedWriter` ajoutent une mémoire tampon au lecteur (resp. écrivain) sous-jacent, améliorant ainsi potentiellement les performances.

De plus, `BufferedReader` ajoute une méthode à `Reader` :

- `String readLine()` : lit et retourne la prochaine ligne du lecteur sous-jacent, ou `null` si la fin a été atteinte,

et `BufferedWriter` ajoute une méthode à `Writer` :

- `void newLine()` : écrit une fin de ligne dans l'écrivain sous-jacent.

6.1.7 Lecteur comptant les lignes

`LineNumberReader` est une sous-classe de `BufferedReader` qui garde trace du numéro de ligne actuel — en commençant à 0 — et permet de le consulter ou modifier au moyen des méthodes suivantes :

- `int getLineNumber()` : retourne le numéro de ligne actuel, et
- `void setLineNumber(int l)` : définit le numéro de ligne à la valeur donnée — sans que cela n'affecte le lecteur sous-jacent de quelque manière que ce soit.

6.1.8 Exemple

L'extrait de programme ci-dessous lit un fichier nommé `utf8.txt` encodé en UTF–8 et l'écrit en UTF–16 dans `utf16.txt` :

```
try (Reader i = new InputStreamReader(
                  new FileInputStream(”utf8.txt”),
                  StandardCharsets.UTF_8);
     Writer o = new OutputStreamWriter(
                  new FileOutputStream(”utf16.txt”),
                  StandardCharsets.UTF_16)) {
  int c;
  while ((c = i.read()) != -1)
    o.write(c);
}
```

7 Références

- la documentation de l'API Java, en particulier les classes et interfaces suivantes :
  - la classe `InputStream` et ses sous-classes,
  - la classe `OutputStream` et ses sous-classes,
  - la classe `Reader` et ses sous-classes,
  - la classe `Writer` et ses sous-classes,
  - l'interface `AutoCloseable`,
  - la classe `StandardCharsets`,
  - la classe `Character`,
  - la classe `String`,
- *The Java® Language Specification*, de James Gosling et coauteurs, en particulier :

– §14.20.3 *try-with-resources*,
– §3.10.4 *Character Literals*,
– §3.10.5 *String literals*.