LES FLUX

D'ENTREE / SORTIE

Les flux

Un **flux** (**stream** en anglais) d'E/S sert à **envoyer** ou **recevoir** des données. C'est donc une **source d'entrée** (`in`) ou une **destination de sortie** (`out`).

Un flux peut avoir différentes natures de sources/destinations (fichiers sur disque, périphériques, autres processus, zones de mémoire,...). Les flux cachent les spécificités de fonctionnement internes et offre à l'utilisateur un même modèle simple : un flux est juste une séquence de données. Un flux traite les données **séquentiellement**.

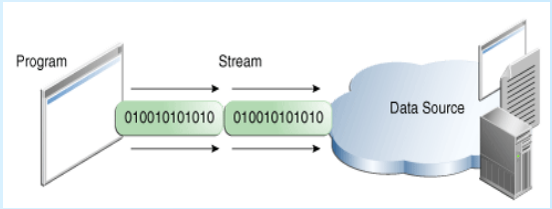
Les flux peuvent véhiculer différents types de données. La plus petite unité de transmission est l'**octet**. Certains flux transmettent juste les données (suite d'octets) ; d'autres les transforment au passage (encodages / décodages). Java distingue :

- les flux d'**octets** (byte) : E/S de bas-niveau
- les flux de **caractères** : gèrent les problèmes d'encodages de caractères (bâtis sur les flux d'octets).

Les flux

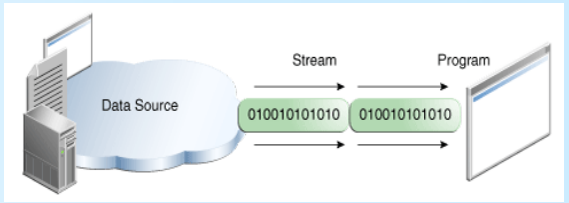
Flux d'entrée (in): flux sur lequel on peut faire des opérations de lecture.
Le schéma du programme est:

ouverture du flux, lectures, …, fermeture



Flux de sortie (out): sur lesquels on peut faire des opérations d'écriture.
Le schéma du programme est:

ouverture du flux, écritures, …, fermeture



L'ouverture se fait en instanciant un objet correspondant à la nature du flux (le constructeur accepte en paramètre la désignation de la source).
Ex: `new FileReader("resultats.txt");`

La fermeture se fait par un appel de méthode `close()` qui permet de libérer les ressources utilisées pour gérer le flux (ex: buffer).

La plupart des opérations peuvent lever des `IOException`.

Les flux : classes racines

Classes racines (classes abstraites):

	Flux d'octets	Flux de caractères
Flux d'entrée	`InputStream`	`Reader`
Flux de sortie	`OutputStream`	`Writer`

Les sous-classes ont des noms de la forme <préfixe><classe racine>.

Classes propres à la nature du flux. Le préfixe indique alors la nature.
Le constructeur permet d'indiquer la source du flux. Ex: pour lire un fichier disque: `FileReader f = new FileReader("resultats.txt");`

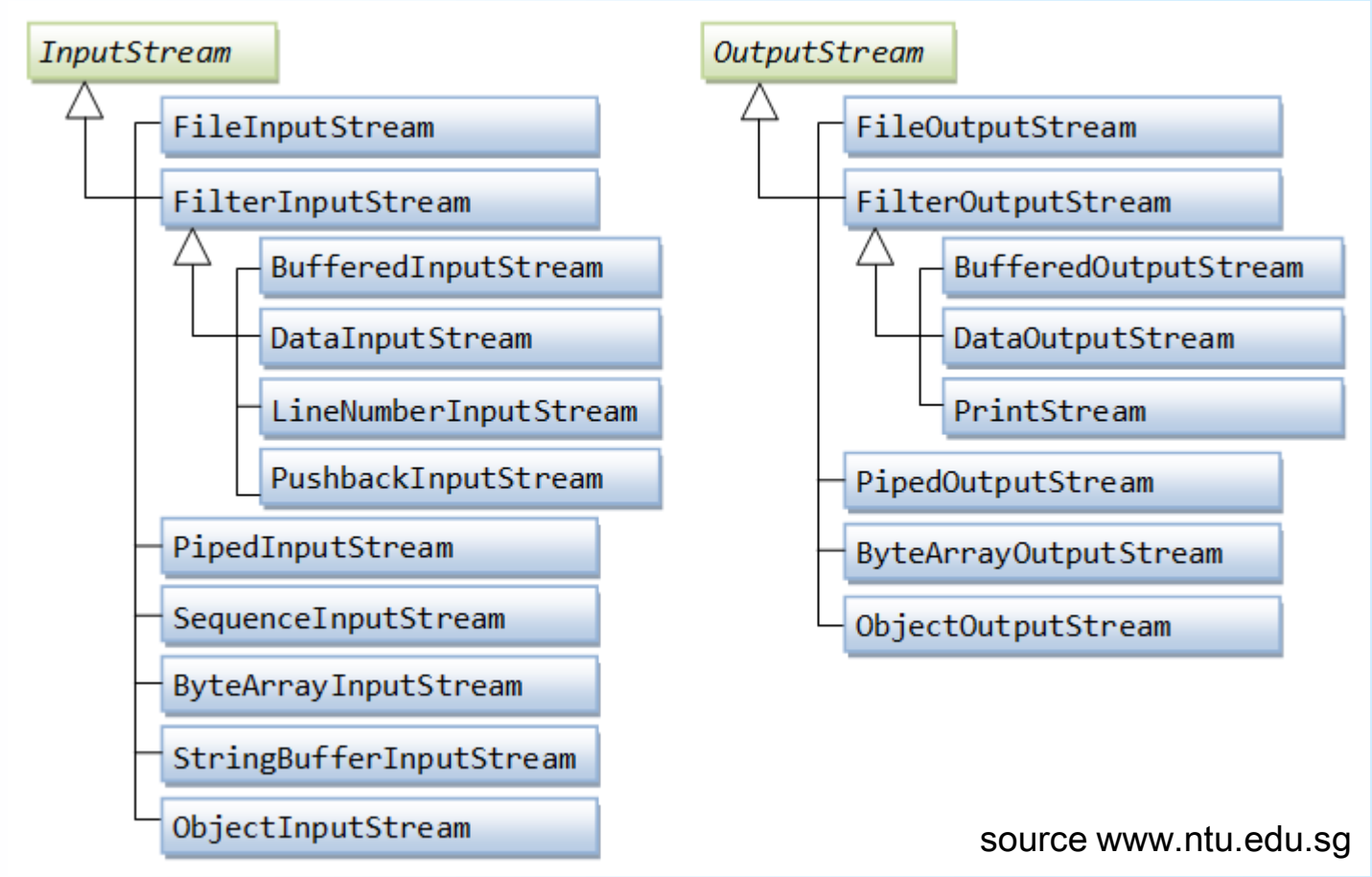
Classes « filtres » : elles s'intercalent entre l'application et un flux pour lui ajouter des fonctionnalités. Le préfixe indique l'action du filtre.
Le constructeur prend le flux à modifier. Ex: lecture bufferisée:

```
BufferedReader f1 = new BufferedReader(f);
```

Les flux : classes principales

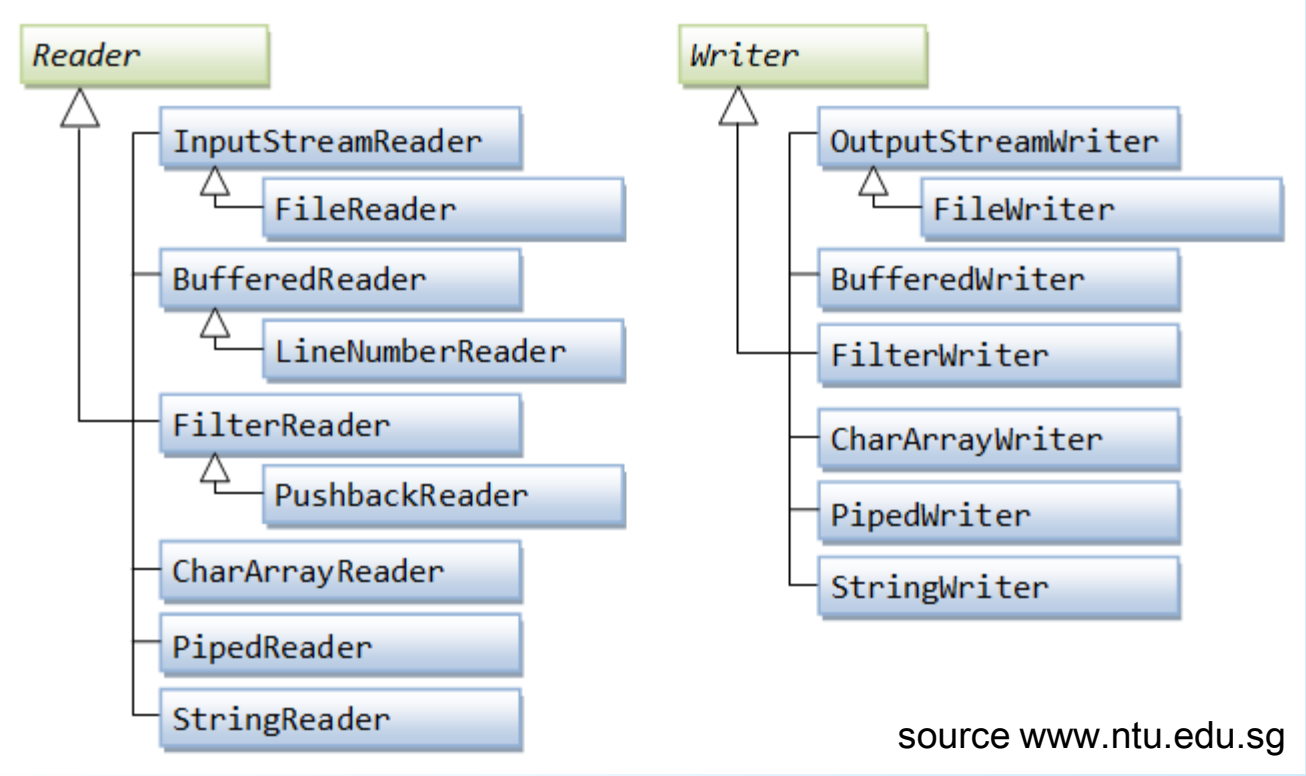
	Nature du flux ou filtre	préfixe	octets / caracs	entrée / sortie
Nature du flux	Fichier disque	**`File`**	les 2	les 2
	Tableau d'octets	**`ByteArray`**	octets	les 2
	Tableau de caractères	**`CharArray`**	caractères	les 2
	Chaîne de caractères	**`String`**	caractères	les 2
opération réalisée par le filtre	Gestion d'un buffer	**`Buffered`**	les 2	les 2
	Numérotation des lignes	**`LineNumber`**	octets	entrée
	Remise dans le flux d'entrée	**`PushBack`**	les 2	entrée
	Formatage de la sortie	**`Print`**	les 2	sortie
	E/S types primitifs	**`Data`**	octets	les 2
	E/S objets (sérialisation)	**`Object`**	octets	les 2
	Lecture octets → caractères Ecriture caractères → octets	**`InputStream`** **`OutputStream`**	octets	entrée sortie
	Séquence de flux	**`Sequence`**	octets	entrée
	« pipe » entre threads	**`Piped`**	les 2	les 2

Hiérarchie pour le flux d'octets



source www.ntu.edu.sg

Hiérarchie pour le flux de caractères



source www.ntu.edu.sg

Flux d'octets en entrée

`InputStream` définit une méthode (abstraite) primitive de lecture:

- **`int read()`**: retourne le prochain octet (entre 0 et 255) ou -1 si EOF.

Ainsi que d'autres fonctions utilitaires :

- **`int read(byte[] b)`**: lit des octets et remplit le tableau **`b`**. Retourne le nombre d'octets lus ou -1 si EOF.
- **`int read(byte[] b, int off, int len)`**: lit au maximum **`len`** octets et les stocke dans **`b`** partir de l'indice **`off`**. Retourne le nombre d'octets lus ou -1 si EOF.
- **`byte[] readAllBytes()`**: lit les octets restants et les retourne dans un tableau nouvellement alloué.
- **`long skip(long n)`**: lit et ignore les n prochains octets. Retourne le nombre effectifs d'octets sautés.

Ces méthodes sont disponibles dans toute sous-classe de **`InputStream`**.

Flux d'octets en sortie

`OutputStream` définit une méthode (abstraite) primitive d'écriture:

- **`void write(int b)`** : écrit le prochain octet b (entre 0 et 255).

Ainsi que d'autres fonctions utilitaires :

- **`void write(byte[] b)`** : écrit les octets du tableau **`b`**.
- **`void write(byte[] b, int off, int len)`** : écrit **`len`** octets du tableau **`b`** à partir de l'indice **`off`**.
- **`byte[] writeAllBytes()`** : écrit les octets restants et les retourne dans un tableau nouvellement alloué.

Ces méthodes sont disponibles dans toute sous-classe de **`OutputStream`**.

E/S d'octets bufferisées

`BufferedInputStream` implémente les méthodes :

- **`void mark(int readAheadLimit)`** : pose une marque pour repositionner le flux avec un reset(). L'entier **`readAheadLimit`** est le nombre maximal d'octets pouvant être lus avant que la marque ne soit plus utilisable.
- **`void reset()`** : repositionne le flux à la dernière marque.

`BufferedOutputStream` implémente les méthodes:

- **`void flush()`** : force l'écriture physique des données (envoie les données actuellement dans le buffer sur le flux).

Exemple : copie de fichier

Exemple de code de copie de fichier:

```
FileInputStream in = new FileInputStream("fichier1.txt");
FileOutputStream out = new FileOutputStream("sauve.txt");
int c;
while((c = in.read()) != -1) {
   out.write(c);
}
in.close();
out.close();
```

Questions ?

- Choix du type de flux (octet / caractère) correct ?
- Gestion des erreurs d'E/S ?
- Les flux sont-ils toujours bien fermés ?

Exemple : fermeture améliorée

```
FileInputStream in = null;
FileOutputStream out = null;
try {
    in = new FileInputStream("fichier1.txt");
    out = new FileOutputStream("sauve.txt");
    int c;
    while((c = in.read()) != -1) {
        out.write(c);
     }
} finally { // NB: un bloc finally est toujours exécuté
    if (in != null)
        in.close();
    if (out != null)
        out.close();
}
```

Question : gestion des erreurs ?

Exemple : fermeture avec try-with-resources

Java 7 introduit une instruction simplifiant la fermeture de flux (et plus généralement la récupération de ressources). Elle se base sur une interface **`AutoCloseable`** qui définit juste une méthode **`close()`**.

```
try(FileInputStream in = new FileInputStream("fichier1.txt");
    FileOutputStream out = new FileOutputStream("sauve.txt")) {
    int c;
    while((c = in.read()) != -1) {
        out.write(c);
    }
} …
```

A la fin du bloc **`try`**, la méthode **`close()`** des 2 flux est automatiquement invoquées (qu'il y ait ou non une exception levée). On peut évidemment capturer l'erreur avec un

```
… catch(IOException ex){
    // NB: gérer l'exception ici
}
```

Question: code efficace ?

Manipulation de caractères

Pour traiter du texte en informatique, il faut le stocker en mémoire sous forme numérique. Le texte étant composé de caractères (lettres, chiffres, ponctuations, symboles) on utilise un codage pour chaque caractère.

Le codage le plus populaire est l'ASCII qui permet de représenter 128 caractères (numérotés de 0 à 127, 00 à 7F). Le 'A' a pour code 65 (41 en hexa). En ASCII, 1 octet suffit pour coder tout caractère (et il reste même 128 codes disponibles.

Le code Latin-1 (ISO-8859-1) est un sur-ensemble de l'ASCII utilisant les 128 autres caractères pour les accents des langues européennes et autres symboles (codes de 128 à 255, 80 à FF).

Comment coder plus de 256 caractères ? En restant sur 1 octet c'est difficile (cf. pages de code, qui utilisent différemment la plage 128 à 255).

Il faut se résoudre à utiliser plusieurs d'octets. Les problèmes de représentations arrivent (combien d'octets ? dans quel ordre ?...).

Manipulation de caractères

On distingue aujourd'hui:

- L'ensemble des caractères pris en charge (le charset)
- La correspondance entre caractères et entiers (points de code)
- Comment les points de code sont codés en numérique sous la forme d'une série d'octets et dans quel ordre les octets apparaissent (l'encodage).

Avec l'ASCII ou le Latin1, le charset était petit, les caractères étaient confondus avec leur point de code, lui-même codé sous la forme d'un seul octet.

On trouve souvent la confusion charset ⇔ encodage (notamment en Java).

L'Unicode

Java utilise l'Unicode. Unicode (www.unicode.org) associe à chaque caractère un entier unique appelé point de code (*code point*) qui s'écrit sous la forme U+XXXX (X comporte de 4 à 6 chiffres héxadécimaux).

Ainsi le 'A' a pour code point U+0041(soit 65 en décimal). Unicode correspond exactement à l'ASCII pour les 127 premiers codes. L'Unicode va plus loin et veut normaliser tous les caractères de toutes les langues (y compris les emoji !).

Unicode utilise 21 bits soit de U+0000. à U+10FFFF (mais des plages sont réservées, ainsi U+D800 à U+DFFF sont des non-caractères). Aujourd'hui environ 260 000 caractères sont référencés. Toutefois, Unicode continue d'évoluer.

Unicode va plus loin: chaque caractère à un nom, comment le convertir en majuscule / minuscule,… Ainsi le **à** s'appelle LATIN SMALL LETTER A WITH ACUTE, code U+00E1, sa majuscule est **Á**, de code U+00C1.

Encodages pour l'Unicode

L'encodage désigne la manière dont un point de code va être stocké sur un support numérique (mémoire, disque,…). Il s'agit de coder un entier en numérique (en base 2). Pb: combien d'octets va-t-on utiliser pour cet encodage ? Théoriquement 32 bits sont nécessaire (UTF-32) mais il est très gourmand en espace mémoire. On a donc inventé des tas d'encodages (trop !). Ex: l'Universal character set Transformation Format:

caractère	code	encodage	octets
A	U+0041	UTF-8	01000001
A	U+0041	UTF-16	00000000 01000001
A	U+0041	UTF-32	00000000 00000000 00000000 01000001
あ	U+3042	UTF-8	11100011 10000001 10000010
あ	U+3042	UTF-16	00110000 01000010
あ	U+3042	UTF-32	00000000 00000000 00110000 01000010

Encodages pour l'Unicode

De plus, dans le cas où l'encodage ne le force pas se pose la question de l'ordre des octets (endianness). Quel sera le premier octet lu ? L'octet de poids faible (little endian) ou l'octet poids fort (big endian) ?

Variantes : UTF-16BE, UTF-16LE. Si rien n'est précisé on regarde les 2 premiers octets du flux (fichier texte), si c'est un BOM (Byte Order Mark) on le respecte sinon on considère que c'est BE. Le BOM est le caractère Unicode U+FEFF , suivant comment il est stocké on peut déterminer l'encodage du fichier.

Un BOM indique qu'on est présence d'un fichier Unicode (UTF-8) et l'endianness (UTF-16, UTF-32). Le BOM n'est pas obligatoire !

Tout cela est bien compliqué, c'est la raison des flux de caractères…

Octets	Encodage
00 00 FE FF	UTF-32, big-endian
FF FE 00 00	UTF-32, little-endian
FE FF	UTF-16, big-endian
FF FE	UTF-16, little-endian
EF BB BF	UTF-8

Java et l'Unicode

Lorsque Java a été crée (1995), l'Unicode tenait sur 16 bits (65536 codes). De ce fait le type primitif **`char`** occupe 16 bits. C'est insuffisant aujourd'hui mais trop tard pour modifier le type **`char`**. Les char sont et restent limités aux caractères U+0000..U+FFFF (appelé BMP *Basic Multilingual Plane*).

Java n'a pas modifié son type char mais a opté pour l'UTF-16.

En UTF-16, un caractère nécessite 1 × 16 bits (pour le BMP) ou 2 × 16 bits pour les caractères entre U+10000..U+10FFFF. Il faut donc 1 ou 2 **`char`** pour représenter un caractère Unicode. Dans le cas de 2 **`char`** on parle de *surrogate pair* (paire de substitution).

Ainsi un **`String`**, **`StringBuffer`** ou un **`char[]`**, peuvent contenir des paires (2 **`char`** successifs). Ca complique le traitement, ex: le **`.length()`** ne donne plus le nombre de caractères mais le nombre de **`char`**…, cf API surrogates, méthodes tq: **`isSurrogatePair()`** ou **`codePointAt()`**.

Confusion Java: que désigne « caractère » : caractère Unicode ou **`char`** ?

Délimiteur de fin de ligne

Dans un flux de caractères (ex: fichier texte), il est souvent important de distinguer les lignes entre elles. Pour cela il faut reconnaître le caractère fin de ligne. Il fait partie des 32 premiers codes ASCII (00 à 1F, soit 00 à 31), appelés codes de contrôles. Malheureusement plusieurs conventions existent… la plupart à base des caractères CR (Carriage Return = 0D = 13 = `'\r'`) et LF (Line Feed = 0A = 10 = `'\n'`). Les plus courants:

Nom	Système	Séquence
CR	Mac OS 9 (vieux)	0D
LF	Unix, Mac OS X (récents)	0A
CRLF	Windows	0D 0A

Java: utiliser **`println`** ou **`System.getProperty("line.separator")`** ou **`%n`** dans **`printf`**.

Conseil: en parsing, ignorer les CR et considérer que LF marque une fin de ligne.

Flux de caractères en entrée

`Reader` définit une méthode (abstraite) primitive de lecture:

- **`int read(char[] c, int off, int len)`**: lit au maximum **`len`** caractères et les stocke dans **`c`** partir de l'indice **`off`**. Retourne le nombre d'caractères lus ou -1 si EOF.

Ainsi que d'autres fonctions utilitaires :

- **`int read()`**: retourne le prochain caractère (char entre U+0000 et U+FFFF) ou -1 si EOF.
- **`int read(char[] c)`**: lit des caractères et remplit le tableau **`c`**. Retourne le nombre d'caractères lus ou -1 si EOF.
- **`long skip(long n)`**: lit et ignore les n prochains caractères. Retourne le nombre effectifs de caractères sautés.

Ces méthodes sont disponibles dans toute sous-classe de **`Reader`**.

Flux de caractères en sortie

`Writer` définit une méthode (abstraite) primitive d'écriture:

- **`void write(char[] c, int off, int len)`**: écrit **`len`** caractères du tableau **`c`** à partir de l'indice **`off`**.

Ainsi que d'autres fonctions utilitaires :

- **`void write(int c)`**: écrit le prochain caractère c (entre U+0000 et U+FFFF).
- **`void write(char[] c)`**: écrit les caractères du tableau **`c`**.
- **`void write(String str)`**: écrit la chaîne de caractères **`str`**.
- **`write(String str, int off, int len)`** : écrit **`len`** caractères de la chaîne **`str`** à partir de l'indice **`off`**.

Ces méthodes sont disponibles dans toute sous-classe de **`Writer`**.

E/S de caractères bufferisées

`BufferedReader` implémente les méthodes :

- **`String readLine()`** : lit la prochaine ligne (jusqu'au délimiteur de fin de ligne).
- **`void mark(int readAheadLimit)`** : pose une marque pour repositionner le flux avec un reset(). L'entier **`readAheadLimit`** est le nombre maximal de caractères pouvant être lus avant que la marque ne soit plus utilisable.
- **`void reset()`** : repositionne le flux à la dernière marque.

`BufferedWriter` implémente les méthodes:

- **`void flush()`** : force l'écriture physique des données (envoie les données actuellement dans le buffer sur le flux).
- **`void newLine()`** : écrit un délimiteur de fin de ligne, obtenu par **`System.getProperty("line.separator")`**

Sorties formatées

`PrintStream` et **`PrintWriter`** permettent de formater les sorties.

Peu de différence entre les 2: **`PrintStream`** utilise l'encodage par défaut de l'architecture alors que **`PrintWriter`** utilise celui précisé pour le flux.

Fournissent les méthodes **`print`**, **`println`** et **`printf`** déjà vues.

Préciser l'encodage

Peu de constructeurs de flux permettent de préciser le charset/encodage à utiliser. Ils est souvent nécessaire de passer par un filtre du type **`InputStreamReader`** / **`OutputStreamWriter`** qui transforme un flux octet en un flux caractère (et permet aussi de préciser un encodage).

Exemple: écriture d'un fichier caractère en précisant latin1:

```
Writer f = new OutputStreamWriter(
                new FileOutputStream("doc.txt"),
                "ISO-8859-1"); // "Latin1" accepté aussi
f.write("élève");
f.close();
```

Voir la classe **`CharSet`** et ses méthodes **`encode()`** et **`decode()`**.