# 

# Les fichiers

# 

## Les flux (Streams)

Un flux est un chemin parcouru par des données à l’intérieur d’un programme.

* *Flux d’entrée* : de la source à un programme.
* *Flux de sortie* : du programme à la destination.
* *Flux d’octets* : ils véhiculent des entiers compris entre 0 et 255. Ils permettent de véhiculer des données numériques, des fichiers exécutables, du pseudo-code et des communications Internet.
* *Flux de caractères* : ne traitent que des données textuelles. Ils supportent le standard Unicode.  
    
  Ex : fichiers textes et pages Web.

## Filtre

Type de flux qui change la gestion d’un autre flux existant. Il s’apparente à un barrage. Étapes à suivre pour utiliser un filtre sur un flux :

* Créer un flux associé à une source ou à une destination de données;
* Associer un filtre à ce flux;
* Lire ou écrire des données à partir du filtre, et non à partir du flux original.

## Flux de fichiers

Les flux d’octets qui seront utilisés seront sans doute les flux de fichiers : ils servent à communiquer des données avec des supports et/ou périphériques servant à stocker des informations. Pour avoir accès à un fichier, il suffit de donner le chemin du dossier et le nom du fichier.

## Flux d’entrée de fichier

La création d’un flux d’entrée de fichier se fait grâce à la méthode constructeur **FileInputStream(String)**. La chaîne de caractères donnée en paramètre doit être le nom du fichier et, s’il y a lieu, également le chemin pour y accéder. Exemple de création d’un flux d’entrée de fichier:

**FileInputStream FluxEntreeFichier = new FileInputStream("Allo.txt");**

Puisque le flux d’entrée est créé, il est maintenant possible de lire des octets qui proviennent de ce flux grâce à la méthode **read()**. Elle retourne un entier qui contient l’octet suivant du flux. Si c’est –1, la fin du fichier a été atteinte. Cette méthode lit un octet à la fois.

Pour lire plus d’un octet, il suffit d’utiliser la méthode **read(byte[], int, int)** :

* Une matrice d’octets pour le stockage des données;
* L’élément de la matrice où doit être stocké le premier octet lu;
* Le nombre d’octets à lire.

Elle retourne le nombre d’octets lus, ou –1 si aucun octet n’a été lu avant la fin du flux. Lorsque le dernier octet a été lu dans le fichier, il s’agit de fermer le flux avec la méthode **close**().Voici un exemple simple pour lire un fichier et afficher son contenu :

**Fichier** : FichierLectureOctets.java (voir le code sur le réseau) Ce code lit octet par octet jusqu’à ce que la fin du fichier (-1) soit atteinte.

## Flux de sortie de fichier

Le constructeur à utiliser est **FileOutputStream(String)**. Le principe est le même qu’avec **FileInputStream**. Pour ajouter des données à la fin d’un fichier existant, il suffit d’utiliser le constructeur **FileOutputStream**(String, **boolean**). String sert au nom (et au chemin s’il y a lieu) du fichier. L’argument « **boolean** » doit être à « **true** » pour que les données s’ajoutent à la fin de celles que contient un fichier existant au lieu de les écraser.

La méthode **write**(int) du flux de sortie de fichier sert à écrire des octets dans le flux. Cette méthode écrit un octet à la fois. Lorsque le dernier octet a été écrit dans le fichier, il s’agit de fermer le flux avec la méthode **close**().

Pour écrire plusieurs octets d’un seul coup, il est possible de se servir de la méthode **write**(byte[], int, int). Le rôle des trois paramètres est le même qu’avec **read**(byte[], int, int) vue précédemment.

**Fichier** : FichierEcritureOctets.java (voir le code sur le réseau)

## 

## Filtrer un flux :

Un flux filtré est un flux qui effectue des changements sur les informations envoyées par l’intermédiaire d’un autre flux déjà existant. Un flux filtré peut se crée grâce aux sous-classes héritières de **FilterInputStream** et **FilterOutputStream**. Ces deux classes ne s’occupent pas de la filtration elle-même; pour que la filtration ait lieu, une de ces deux classes par exemple, doit être utilisée : **DataInputStream** ou **DataOutputStream**. Voir l’API de Java.

## Filtres d’octets

Il est plus efficace d’envoyer des gros volumes d’informations (paquets d’info) que par petits bouts (octet par octet, par exemple), même si elles ne sont pas toutes traitées en même temps. En effet, il est plus rapide de se rendre à un endroit adapté au va-et-vient (tampon ou « buffer », en mémoire) que de se déplacer jusqu’à l’emplacement source de ces informations. Il y a alors un gain de temps non négligeable, le programme n’en sera que plus efficace.

## Flux à tampon

Un flux à tampon place des données qui n’ont pas encore été traitées par le programme, dans un tampon de données (buffer). Elles serviront lorsque le programme en aura besoin, car il ira d’abord regarder dans ce tampon avant d’aller à la source originale du flux. Les flux d’octets à tampon utilisent les classes **BufferedInputStream** et **BufferedOutputStream**.

Constructeurs de **BufferedInputStream**:

//créer un flux d’entrée à tampon pour un objet InputStream

//donné en paramètre.

**BufferedInputStream(InputStream)**

//donne en plus la taille « int » octets au buffer créé.

**BufferedInputStream(InputStream, int)**

Les méthodes **read**() citées plus tôt sont disponibles pour le flux à tampon.

Constructeurs de **BufferedOutputStream**:

//créer un flux d’entrée à tampon pour un objet OutputStream

//donné en paramètre

**BufferedOutputStream(OutputStream)**

//donne en plus la taille int octets au buffer créé

**BufferedOutputStream(OutputStream, int)**

Les méthodes **write**(int) citées plus tôt sont disponibles pour le flux à tampon.

Malgré le fait que le paramètre de **write**(int) soit un int, la valeur doit être comprise entre 0 et 255. Si un nombre est 256 ou +, le programme stockera le reste de la division (opérateur modulo %).

**Fichier** : FichierLectureBufferOctets.java (voir le code sur le réseau)

**Fichier** : FichierEcritureBufferOctets.java (voir le code sur le réseau)

## Flux de données

Pour les données qui sont représentées autrement qu’en octets et qu’en caractères (types de données primitives : int, float…). Les données sont considérées dans le type auquel elles appartiennent. Dans le cas des données float, elles sont écrites comme float et sont lues en tant que float. Il y a 3 étapes à respecter pour utiliser un flux de données en bonne et due forme :

* **Créer un flux d’entrée ou de sortie associé à un fichier;**
* **Associer un nouveau flux d’entrée ou de sortie à tampon au flux de fichier nouvellement créé;**
* **Associer un nouveau flux spécialisé d’entrée ou de sortie de données au flux à tampon.**

Exemple pour **écrire** des données typées sur le disque:

//Associer un flux de sortie à un fichier

FileOutputStream **fluxSortieFichier** = new FileOutputStream("Donnees.txt");

//associer un flux de sortie à tampon à un flux de fichier

BufferedOutputStream **fluxATampon** = new BufferedOutputStream(**fluxSortieFichier**);

//associer un flux de sortie de données à un flux de sortie à tampon

DataOutputStream fluxSortieDonnees = new DataOutputStream(**fluxATampon**);

Le même principe est utilisé pour la **lecture** des données :

//associer un flux d’entrée à un fichier

**FileInputStream fluxEntreeFichier = new FileInputStream("Donnees.txt");**

//associer un flux d’entrée à tampon à un flux d’entrée du fichier

**BufferedInputStream fluxATampon = new BufferedInputStream(fluxEntreeFichier);**

//associer un flux d’entrée de données spécialisées au un flux d’entrée à  
//tampon

**DataInputStream fluxEntreeDonnees = new DataInputStream(fluxATampon);**

Méthodes de **lecture** et **d’écriture** pour les différents types de données sur les 2 « data stream » :

readBoolean(), writeBoolean(boolean)

readByte(), writeByte(integer)

readDouble(), writeDouble(double)

readFloat(), writeFloat(float)

readInt(), writeInt(int)

readLong(), writeLong(long)

readShort(), writeShort(int)

readUTF(), writeUTF(String)

// Écriture d’un booléen

fluxSortieDonnees.writeBoolean(ok); //écrire une valeur booléenne

// Lecture d’un booléen

ok = fluxEntreeDonnees.readBoolean(); //recevoir une valeur booléenne

Les méthodes : readUnsignedByte() et readUnsignedShort() lisent des octets et des données short sans signe. Cependant, comme il n’y a pas de « unsigned » en Java, ils sont retournés sous forme d’entiers « int », compatibilité C, C++.

**Les méthodes de lecture de flux d’entrée de données ne sont pas toutes en mesure de renvoyer une valeur signifiant la fin d’un flux. Une manière de contourner ce « risque » est d’attendre qu’une exception EOFException soit lancée. Il suffit de faire la lecture du flux à l’intérieur d’un bloc « try » (car elle est susceptible de générer une exception). L’exception sera évidemment attrapée dans un bloc catch, qui servira à fermer le flux.**

**Fichier** : FichierDataInputOutput.java (voir le code sur le réseau)

### 

## Flux de caractères :

Les flux de caractères permettent de travailler avec des fichiers de texte, comme des documents HTML et des fichiers sources de programmes. Au lieu d’utiliser directement les flux d’octets, il faut utiliser les classes qui héritent de **Reader** et **Writer**. Voir l’API de Java

## Lire des fichiers de texte

L’association d’un fichier à un flux d’entrée de caractères se fait à l’aide du constructeur **FileReader**(String) où **String** est le nom du fichier et son chemin s’il y a lieu.

FileReader lireFichier = new **FileReader**("Agenda.c");

Méthodes disponibles :

**read()** : renvoie le caractère qui suit dans le flux sous le type int;

**read(char[], int, int)** : Place les caractères lus dans le vecteur de caractères à partir de l’indice spécifié par le paramètre du centre jusqu’à ce que le nombre spécifié par l’argument de droite soit atteint.

read(vecCar, 5, 8); //met les caractères du vecteur de

//caractères à partir de matCar[5] et

//continue jusqu’à ce qu’il en ait lu 8

//(matCar[12] inclus).

Au lieu de retourner le caractère qui suit, cette méthode retourne les caractères lus si tout va bien, ou –1 s’il n’y en a aucun (fin du flux).

## Écrire des fichiers de texte :

L’écriture d’un flux de caractères dans un fichier se fait grâce à la classe FileWriter.

Elle offre deux constructeurs :

**FileWriter(String)**

**FileWriter(String, boolean)**

Le booléen est facultatif. Il est à « true » pour ajouter les données à la fin du fichier, « false » pour créer ce fichier (ou l’écraser s’il est déjà existant). L’autre argument est un String pour indiquer le nom du fichier et son chemin de dossier, s’il y a lieu.

Trois méthodes :

**write(int)** : permet l’écriture d’un caractère (fourni en paramètre en tant qu’entier).

**write(char[], int, int)** : ces arguments ont le même rôle que précédemment (voir « lire des fichiers de texte »).

**write(String, int, int)** : Écrit des caractères de la chaîne envoyée en paramètre, à partir du point de départ indiqué par le premier argument int et en écrivant le nombre de caractères indiqué par le second.

// Exemple

FileWriter chaineCar = FileWriter("writeString.txt");

chaineCar.write("abcdefghij", 2, 7);

chaineCar.**close()**;

Exemple utilisant à la fois **FileReader** et **FileWriter** :

**Fichier** : FichierCaractereInputOutput.java (voir le code sur le réseau)

## Buffers (tampons) de flux de caractères

Il est également possible d’utiliser un buffer pour pouvoir lire plus d’un caractère à la fois. La classe qui permet d’obtenir ce tampon est **BufferedReader**. Elle offre ces deux constructeurs :

**BufferedReader(Reader)** : pour créer un flux de caractères à tampon associé à l’objet envoyé en paramètre.

**BufferedReader(Reader, int)**: permet en plus de fixer la taille de ce tampon avec un paramètre entier.

**Argument Reader** : n’importe quelle classe de flux d’entrée de caractères, comme FileReader. Les méthodes **read**() et **read**(char[], int, int) sont également disponibles (leurs paramètres ont le même rôle que ceux des méthodes de **FileReader** possédant le même nom).

Pour lire les lignes de texte une par une, il suffit d’utiliser la méthode **readLine**(). Elle retourne une chaîne de caractères (String) contenant la ligne suivante du flux, caractère(s) de fin de ligne non inclus. Lorsque la fin du flux est atteinte, la valeur de la chaîne renvoyée est égale à null.

**Fichier** : FichierTexteInputOutput.java (voir le code sur le réseau)

La classe **BufferedWriter** peut servir pour écrire un flux de caractères à tampon. Voici ses constructeurs :

**BufferedWriter(Writer)**

**BufferedWriter(Writer, int)**

**Argument Writer :** n’importe quel objet de flux de sortie de caractères, comme **FileWriter**.

Cette classe possède également les mêmes méthodes de sortie que **FileWriter** : **write**(), **write**(char[], int, int) et **write**(String, int, int).

Il existe également une méthode de sortie qui retourne le caractère de ligne privilégié par la plate-forme utilisée à l’exécution du programme : il s’agit de **newLine()**.

**IMPORTANT** : La méthode **close**() permet la fermeture du flux de caractères à tampon et l’envoi des données du tampon à la destination du flux.

**Fichier** : FichierTexteInputOutput.java (voir le code sur le réseau)

## Classe PrintWriter :

Cette classe qui est aussi héritière de Writer permet d’écrire une ligne de texte dans un fichier. Ce principe ressemble à celui de la méthode readLine() citée plus tôt (ligne par ligne). Il suffit de créer un objet de la classe PrintWriter pour avoir accès à la méthode **print()** ou **println()**, qui permet d’écrire dans un fichier de cette manière.

PrintWriter ecrireLigne = new PrintWriter(fluxSortie);

ecrireLigne.println("Texte à écrire sur la prochaine ligne du fichier");

ecrireLigne.flush();

Voir le document « 1. Entrées et Sorties (IO).pdf » pour plus de détails…

## Fichiers et filtres de noms de fichiers :

La classe **File** permet d’effectuer des opérations sur le fichier lui-même et non son contenu: par exemple, il est possible d’en faire une copie et/ou de le renommer.

Trois constructeurs sont disponibles :

**File(String)** : création d’un objet File avec le dossier seulement, le nom du fichier n’est pas spécifié, ce qui signifie que l’objet ne fait référence qu’au dossier de fichier.

**File(String, String)** : crée un objet File avec le chemin de dossier et le nom du fichier spécifié.

**File(File, String)** : crée un objet File dont le chemin est représenté par File et dont le nom du fichier est représenté par String.

N.B. : Pour créer les flux et leurs tampons (si désirés), il suffit de choisir les constructeurs appropriés. Il faut déterminer le type de fichier et le type de traitement voulu pour le(s) fichier(s) concerné(s).

Principales méthodes disponibles pour les objets **File** :

* exists() : pour savoir si le fichier existe à l’endroit (chemin) spécifié. Une valeur booléenne est retournée.
* length() : retourne un entier long permettant de savoir la taille du fichier en octets.
* renameTo(File) : renommer le fichier. Retourne « true » si tout s’est bien passé.
* delete() / deleteOnExit() : pour supprimer un fichier ou un dossier. La méthode delete() tente de faire la suppression immédiatement, une valeur booléenne est retournée pour indiquer si l’opération s’est bien passée. La méthode deleteOnExit() attend la fin d’exécution du programme pour effectuer la suppression, elle est de type void (ne retourne rien).
* mkdir() : permet la création d’un dossier spécifié par l’objet File sur laquelle la méthode est appelée, elle retourne un booléen.
* getName() : retourne le nom du fichier (en String).
* getPath() : donne seulement le chemin du dossier du fichier.
* getAbsolutePath() : donne le chemin de dossier absolu.
* getParent () : retourne le nom du répertoire parent.
* canRead() : retourne un booléen, qui sert à savoir si le fichier est lisible.
* canWrite() : retourne aussi un booléen, qui permet de savoir si le fichier peut être modifié.
* List() : retourne une matrice de String. Pour obtenir la liste des fichiers du répertoire.

Petit exemple d’application utilisant quelques unes de ces méthodes:

//objet File

**File** f = new **File**("fichierTest.txt");

//affiche le chemin absolu + le nom du fichier

System.out.println(f.**getAbsolutePath**() + f.**getName**());

//si le fichier existe, on peut en savoir plus

if (f.**exists**())

{

//afficher le nom du fichier, s’il est lisible (r) ou non (-) ,

//s’il est modifiable (w) ou non(-) et sa taille en octets.

System.out.println(f.**getName**() + " : " + (f.**canRead**() ? "r" : "-") +

(f.**canWrite**() ? "w" : "-") + " : " + f.**length**() );

//supprimer le fichier

f.**delete**();

}

Toutes ces méthodes sont susceptibles de générer une exception **SecurityException**. Il faut donc prévoir l’utilisation des blocs **try** et **catch** ou de la clause **throws**.