Cours à distance

Les flux d'entrée et sortie

En java, toutes les entrées/sorties sont gérées par des flux (*streams*) : lecture du clavier, affichage sur la console, lecture/écriture dans un fichier, échange de données réseau avec des sockets. Un flux est une série d'informations envoyée sur un canal de communication entre deux entités.

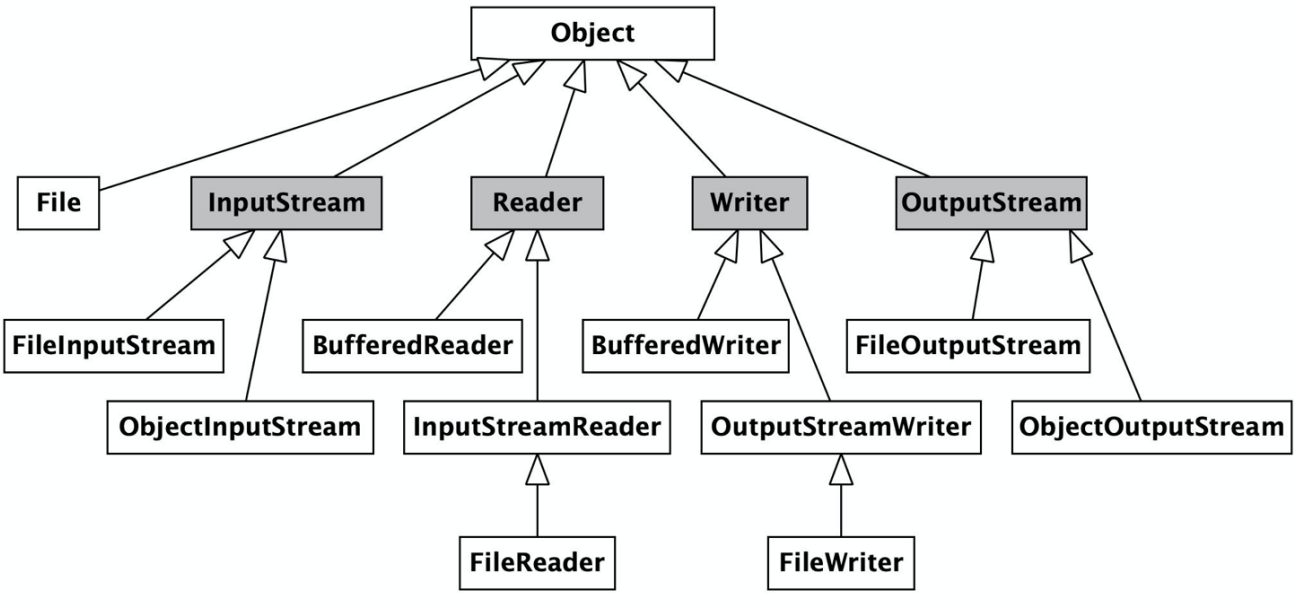
Dans un système d'exploitation, il existe trois flux standards que nous avons déjà vu :

- entre l'application et l'écran pour transmettre une information (`System.out`)
- entre l'application et l'écran pour indiquer une erreur (`System.err`)
- entre le clavier et l'application (`System.in`)

Outre les flux standards, les flux peuvent être divisés en plusieurs catégories :

- les flux d'entrée (*input stream*) et les flux de sortie (*output stream*)
- les flux de traitement de caractères et les flux de traitement d'octets

Java définit des flux pour lire ou écrire des données mais aussi des classes qui permettent de faire des traitements sur les données du flux. Ces classes doivent être associées à un flux de lecture ou d'écriture et sont considérées comme des filtres. Par exemple, il existe des filtres qui permettent de mettre les données traitées dans un tampon (*buffer*) pour les traiter par lots. Toutes ces classes sont regroupées dans le package `java.io` dont voici une partie ci-dessous.



Nous allons étudier ce package afin de connaître son contenu et de savoir s'en servir.

1 Gestion des fichiers : la classe `File`

Un fichier en *Java* est un objet instance de la classe `File`. Cet objet peut représenter indifféremment un fichier ou un répertoire, avec ou sans chemin d'accès, relatif ou absolu. Il est important de noter que cette instance est une notion abstraite, qui ne représente pas nécessairement un fichier réel existant.

Il existe plusieurs constructeurs à cette classe, en voici quelques uns :

- `public File(String name)` : prend en paramètre une chaîne de caractères qui indique un chemin, relatif ou absolu, vers un fichier ou un répertoire.
- `public File(String path, String name)` : prend deux chaînes de caractères en paramètre. La première indique le chemin vers le fichier ou le répertoire. La seconde le nom de ce fichier ou répertoire.
- `public File(File dir, String name)` : ce constructeur est analogue au précédent, sauf que le chemin est exprimé sous la forme d'une instance de `File`.

Cette classe possède des méthodes qui permettent d'interroger ou d'agir sur le système de gestion de fichiers du système d'exploitation, afin de savoir entre autres si ce fichier existe, s'il s'agit d'un répertoire, si l'on peut le lire, etc... dont voici un extrait :

- `public boolean canRead()` : indique si le fichier peut être lu.
- `public boolean canWrite()` : indique si le fichier peut être modifié.
- `public boolean createNewFile()` : crée un nouveau fichier vide.
- `public boolean delete()` : détruit le fichier ou le répertoire. Le booléen indique le succès de l'opération.
- `public boolean exists()` : indique si le fichier existe physiquement.
- `public String getAbsolutePath()` : renvoie le chemin absolu du fichier.
- `public String getName()` : renvoie le nom du fichier ou du répertoire.
- `public String getPath()` : renvoie le chemin du fichier.
- `public boolean isAbsolute()` : indique si le chemin est absolu.
- `public boolean isDirectory()` : indique si l'objet est un répertoire.
- `public boolean isFile()` : indique si l'objet est un fichier.
- `public String[] list()` : renvoie la liste des fichiers et répertoires contenus dans le répertoire.
- `public File[] listFiles()` : renvoie la liste des fichiers et répertoires contenus dans le répertoire sous la forme d'un tableau d'instances de `File`.
- `public static File[] listRoots()` : renvoie la liste des éléments racine du système de fichier sur lequel on se trouve.
- `public boolean mkdir()` : crée le répertoire.
- `public boolean mkdirs()` : crée le répertoire avec création des répertoires manquants dans l'arborescence du chemin.
- `public boolean renameTo()` : renomme le fichier.

Vous les retrouverez dans la documentation de l'API *Java* (`http://docs.oracle.com/javase/8/docs/api/`), il vous faudra la consulter et voici un exemple d'utilisation que vous pouvez tester.

```java
import java.io.File;
public class TestFile {
   public static void main(String[] args) {
      File f = new File("test.txt");
      System.out.println("Est-ce␣qu'il␣existe␣?␣" + f.exists());
      System.out.println("Est-ce␣un␣fichier␣?␣" + f.isFile());
      System.out.println("Affichage␣des␣lecteurs␣a␣la␣racine␣du␣PC␣:␣");
      for(File file : f.listRoots()){
         // Chemin absolu du fichier
         System.out.println(file.getAbsolutePath());
         //On parcourt la liste des fichiers et repertoires
         for(File nom : file.listFiles()){
            //S'il s'agit d'un dossier, on ajoute un "/"
            System.out.println("\t\t" + nom.getName() + ((nom.
               isDirectory()) ? "/" : ""));
         }
         System.out.println("␣");
      }
   }
}
```

2 Les flux (*stream*)

Les flux permettent d'encapsuler les processus d'envoi et de réception de données, c'est en quelque sorte un canal dans lequel de l'information transite. Les flux traitent toujours les données de façon séquentielle, c'est-à-dire dans l'ordre dans lequel l'information y est transmise. Certains flux de données peuvent être associés à des ressources qui fournissent ou reçoivent des données comme : les fichiers, les tableaux d'objets, ... De plus les flux peuvent transporter des octets ou des caractères et être d'entrée (lecture) ou de sortie (écriture).

Toutes les classes de flux sont regroupées dans le package `java.io`. Ce qui déroute dans l'utilisation de ces classes, c'est leur nombre et la difficulté de choisir celle qui convient le mieux en fonction des besoins. Pour faciliter ce choix, il faut comprendre la dénomination des classes : cela permet de sélectionner la ou les classes adaptées aux traitements à réaliser.

Le nom des classes se compose d'un préfixe et d'un suffixe. Il y a quatre suffixes possibles en fonction du type de flux (flux d'octets ou de caractères) et du sens du flux (entrée ou sortie).

Type / Direction	Flux d'**octets**	Flux de **caractères**
Flux d'**entrée**	`InputStream`	`Reader`
Flux de **sortie**	`OutputStream`	`Writer`

Les flux transportent donc des byte ou des char dans une direction (entrée ou sortie) **et d'une source** : ici nous nous interéssons uniquement au **fichier ou objet**. Le nom du type de flux permet de connaître ses caractéristiques. Voici les mots clés associés à des exemples :

Source / Direction	Fichier (vers / à partir) **File**	Objets (envoyer / recevoir) **Object**
Entrée : **In**	**`FileIn`**`putStream`	**`ObjectIn`**`putStream`
Sortie : **Out**	**`FileOut`**`putStream`	**`ObjectOut`**`putStream`

3 Les filtres

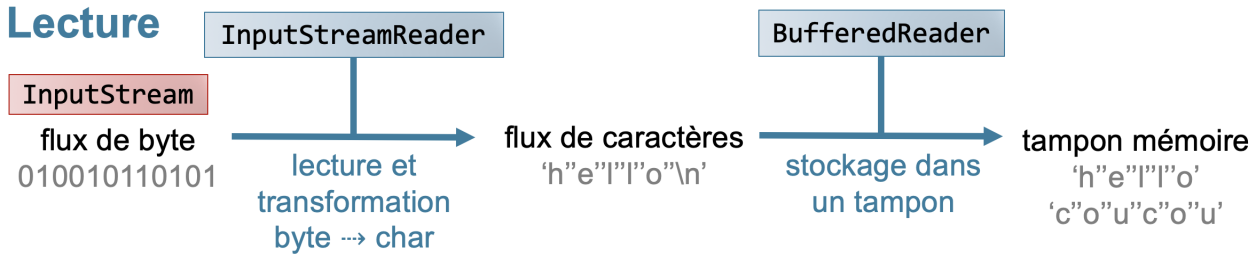
Les filtres sont les différents traitements que l'on peut appliquer sur les flux. Là aussi le préfixe contient le type de traitement qu'il effectue. Attention les filtres n'existent pas obligatoirement pour des flux en entrée et en sortie.

— **Buffered** : ce type de filtre permet de mettre les données du flux dans une mémoire tampon. Il peut être utilisé en entrée et en sortie.
— **Sequence** : ce filtre permet de fusionner plusieurs flux.
— **Data** : ce type de flux permet de traiter les octets sous forme de type de données.
— **LineNumber** : ce filtre permet de numéroter les lignes contenues dans le flux.
— **PushBack** : ce filtre permet de remettre des données lues dans le flux.
— **Print** : ce filtre permet de réaliser des impressions formatées.
— **Object** : ce filtre est utilisé par la sérialisation (*cf. plus loin*).
— **InputStream / OuputStream** : ce filtre permet de convertir des octets en caractères.

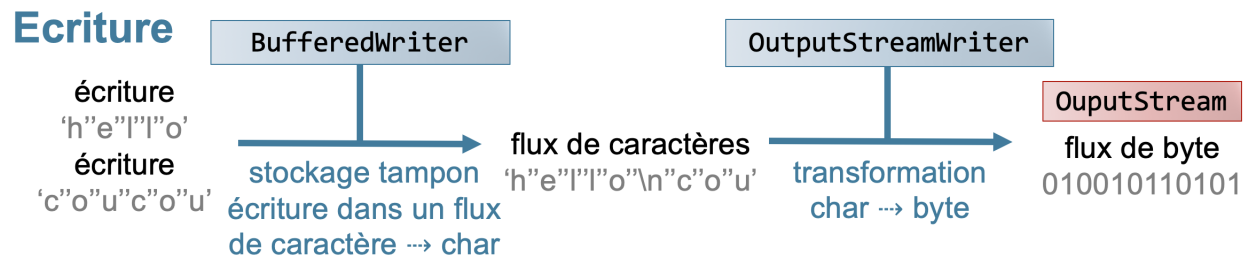
4 Processus de lecture et d'écriture

Le processus de lecture et d'écriture se déroule en 3 phases :
— Ouverture du flux
— Lecture/écriture à partir/vers le flux
— Fermeture du flux

Voici un exemple de processus de lecture (`In`) à partir d'un flux d'octets (`InputStream`) permettant la transformation des bytes en caractères (`InputStreamReader`) afin de la placer dans la mémoire tampon (`BufferedReader`) :



Voici un exemple de processus d'écriture (`Out`) à partir d'un flux de caractères (`Writer`) venant de la mémoire tampon (`BufferedWriter`) permettant la transformation des caractères en bytes (`OutputStreamWriter`) afin de les envoyer vers un flux d'octets (`OuputStream`) :



4.1 Les classes de base

Regardons de plus près les classes de bases : `InputStream`, `OutputStream`, `Reader` et `Writer`. Ce sont des classes abstraites qui définissent les méthodes que chacune des filles devra implémenter pour sa spécificité.

4.1.1 La classe abstraite `InputStream`

Elle correspond à un flux d'entrée d'octets. Voici ces principales méthodes :

- `abstract int read() throws IOException` : lit un octet et le retourne ou -1 si la fin de la source de données est atteinte. C'est cette méthode qui doit être définie dans les sous-classes concrètes et qui est utilisée par les autres méthodes définies dans la classe `InputStream`.
- `int read(byte[] b)` : remplit un tableau d'octets et retourne le nombre d'octets lus.
- `int read(byte[] b, int off, int len)` : remplit un tableau d'octets à partir d 'une position donnée et sur une longueur donnée.
- `void close()` : Permet de fermer un flux. Il faut fermer les flux dès qu'on a fini de les utiliser.

4.1.2 La classe abstraite `Reader`

Elle correspond à un flux d'entrée de caractères codés en Unicode. Voici ces principales méthodes :

- `int read() throws IOException` : lit un caractère, et le retourne sous la forme d'un entier compris entre 0 et 65535 (`0x00-0xffff`), ou -1 si la fin du flux a été atteinte.
- `abstract int read(char[] cbuf) throws IOException` : remplit un tableau de caractères et retourne le nombre de caractères lus.
- `abstract void close()` : permet de fermer un flux. Il faut fermer les flux dès qu'on a fini de les utiliser.

Méthodes des classes dérivées comme `BufferedReader` :

- `String readLine() throws IOException` : lit une ligne de texte et la retourne sous forme de chaîne de caractères.

4.1.3 La classe abstraite `OutputStream`

Elle correspond à un flux de sortie d'octets. Voici ces principales méthodes :

- `abstract void write(int b) throws IOException` : écrit l'octet passé en paramètre.
- `void write(byte[] b) throws IOException` : écrit les octets lus depuis un tableau d'octets.
- `void write(byte[] b, int off, int len) throws IOException` : écrit les octets lus depuis un tableau d'octets à partir d'une position donnée et sur une longueur donnée.
- `void close() throws IOException` : permet de fermer le flux après avoir éventuellement vidé le tampon de sortie.

4.1.4 La classe abstraite `Writer`

Elle correspond à un flux de sortie de caractères codés en Unicode. Voici ces principales méthodes :

- `void write(String str) throws IOException` : écrit une chaîne de caractères.
- `void write(char[] cbuf) throws IOException` : écrit les caractères d'un tableau.
- `abstract void close()` : permet de fermer un flux.

Méthodes des classes dérivées comme `BufferedWriter` :

- `String newLine() throws IOException` : écrit un séparateur de ligne.

4.2 Exemple de lecture

Exemple de lecture de fichier texte : Permet de lire un fichier dont le nom (un objet String) est dans la variable nomFichier.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.io.FileNotFoundException;

class ExempleLecture {
   public static void main (String [] args) {
      // le fichier a lire
      String nomFichier = "ex-fichier.txt";
      String ligne;
      try {
         // creation d'un flux d'entree memoire tampon de caracteres
         // a partir d'un flux d'entree de fichier de caracteres
         BufferedReader in = new BufferedReader(new FileReader(
            nomFichier));
         // lecture de toutes les lignes dans le flux
         while ((ligne = in.readLine()) != null) {
            // affiche la ligne lue
            System.out.println(ligne);
         }
         // fermeture du flux
         in.close();
      }catch (FileNotFoundException e) {
         System.out.println("Probleme de fichier avec " + nomFichier);
      }catch (IOException e) {
         System.out.println("Probleme de lecture : " + e.getMessage());
      }
   }
}
```

La plupart des méthodes concernant les flux propagent des exceptions que nous sommes donc obligés de traiter.

4.3 Exemple d'écriture

Exemple d'écriture de fichier texte : Permet d'écrire dans un fichier dont le nom (un objet String) est dans la variable nomFichier. Si le fichier existe déjà, le texte sera rajoute à la fin, sinon le fichier sera crée et le texte ajouté.

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

class ExempleEcriture{
   public static void main(String[] args){
      // le fichier dans lequel on veut ecrire
      String nomFichier = "nouveauFichier.txt";
      int entier = 5;
      try{
          // creation d'un flux de sortie memoire tampon de caracteres
         // a partir d'un flux de sortie vers un fichier de caracteres
         BufferedWriter out = new BufferedWriter(new FileWriter(
            nomFichier, true));
         // ecriture dans le flux
         out.write("Ajout␣de␣texte␣dans␣un␣fichier\n");
         out.write("On␣peut␣mettre␣des␣entiers␣:␣");
         out.write(entier);
         out.newLine();
         out.write("On␣peut␣mettre␣des␣instances␣de␣Object␣:␣");
         out.write(new Integer(36));
         out.newLine();
         out.write("Attention␣c'est␣la␣description␣de␣l'objet␣qui␣est␣
            ecrite␣(methode␣toString)\n");
         // fermeture du flux
         out.close();
      }catch (IOException e) {
         System.out.println("Probleme␣de␣fichier␣:␣" + e.getMessage());
      }
   }
}
```

5 Lecture / Ecriture d'objet : la sérialisation

La sérialisation d'un objet consiste à le convertir en un tableau d'octets, que l'on peut ensuite écrire dans un fichier, envoyer sur un réseau... Pour cela il suffit de passer tout objet qui implémente l'interface `Serializable` à une instance de flux d'objet (par exemple : `ObjectOutputStream`) pour sérialiser un objet. L'interface `Serializable` n'a pas de méthode, implémenter cette interface consiste donc juste à déclarer cette implémentation.

Des problèmes peuvent se poser pour les objets qui possèdent des attributs qui sont eux-mêmes des objets d'autres classes. Si ces autres classes sont sérialisables, pas de soucis. Sinon ces attributs doivent être marqués avec le mot-clé `transient`. Cela a pour effet de les retirer du flux sérialisé, ces attributs seront donc `null` lors de la lecture de l'objet.

Exemple de lecture et d'écriture d'objet dans un fichier texte :
Attention ceci est juste un extrait, le code est incomplet et ne gère pas les exceptions.

```
Personne pers = new Personne("Doe", "John", 1.75);

// creation d'un flux de sortie d'octets d'objets vers un fichier
ObjectOutputStream oos = new ObjectOutputStream(
                            new FileOutputStream(
                               new File("personne.txt")));
// ecriture de l'objet pers de type Personne
oos.writeObject(pers);
// fermeture du flux
oos.close();

// creation d'un flux d'entree d'octets d'objets a partir d'un fichier
ObjectInputStream ois = new ObjectInputStream(
                            new FileInputStream(
                               new File("personne.txt");
// lecture d'un objet Personne
Personne copiePers = (Personne) ois.readObject();

// comparaison de l'objet avant et apres
if(pers.equals(copiePers))
        System.out.println("cool");
else
        System.out.println("pas cool");
// fermeture du flux
ois.close();
```

6 La classe `Scanner`

Jusqu'ici, nous utilisions la classe `Scanner` simplement pour se brancher à l'entrée standard `System.in` qui est en faite une instance de `InputStream`! En fait, cette classe peut être utilisée avec n'importe quel flux (même un objet `String` et facilite la lecture dans un flux. De plus, elle apporte des fonctionnalités très intéressantes : parser des chaînes de caractères, extraire et convertir les composants.

Cette classe possède plusieurs méthodes `nextXXX()` et `hasNextXXX()` où `XXX` représente un type primitif. Ces méthodes bloquent l'exécution jusqu'à la lecture de données et tente de les convertir dans le type `XXX`. Voici leur fonctionnement :

- Découpe la chaîne de caractères en *tokens* grâce à un délimiteur
- Délimiteur : par défaut un caractère "blanc" (espace, tabulation, retour à la ligne...), mais modifiable via la méthode `useDelimiter(expression)`
- Pour parcourir, récupérer et convertir ces *tokens*

Les méthodes de type `hasNextXXX()` retournent vrai si le prochain *token* complet correspond au type spécifié.

Les méthodes de type `nextXXX()` trouvent et retournent le prochain *token* correspondant au type spécifié.

N'hésitez pas à consulter la documentation de l'API *Java* (`http://docs.oracle.com/javase/8/docs/api/`) pour plus d'informations sur tout ce qui est décrit dans cette fiche.