

JAVA PROGRAMMATION OBJET



**Centre BTS AI KENDI
2DSI**



7. Les Flux

Les Entrées et Les Sorties



DÉFINITION

□ Une **entrée/sortie** en Java consiste en un échange de données entre le **programme** et une **autre source**, qui peut être :

- la mémoire ;
- un fichier ;
- le programme lui-même ;
-

□ En Java, toutes ces données sont échangées **en entrée** et **en sortie** à travers des **flux (Stream)** : c'est un de médiateur entre la **source** des données et sa **destination**.

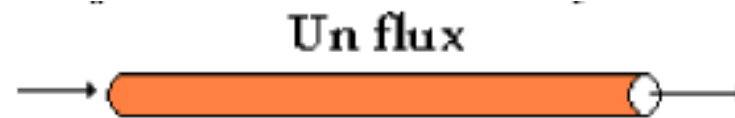
□ Java a décomposé les objets traitant des flux en deux catégories :

- les objets travaillant avec des **flux d'entrée** (in), **lecture de flux** ;
- les objets travaillant avec des **flux de sortie** (out), **écriture de flux**.

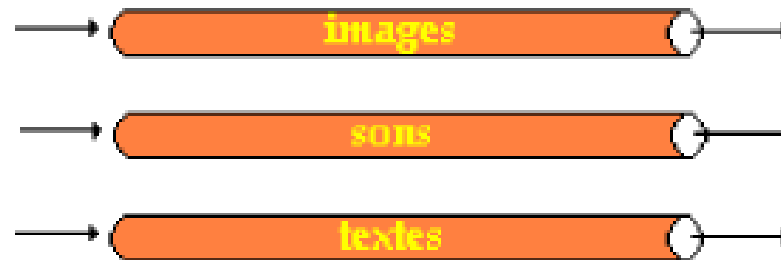


FLUX

Un flux est une sorte de tuyau de transport séquentiel de données



Il existe un flux par type de données à transporter :



Un flux est **unidirectionnel** : il y a donc des **flux d'entrée** et des **flux de sortie** :





FLUX

Flux bufféré

Un flux bufféré stocke les données dans un tampon (buffer) en mémoire centrale, puis lorsque le tampon est plein, le flux transfère le paquet de données contenu dans le tampon vers le fichier (en sortie) ou en provenance du fichier en entrée

Exemple écriture bufférée de caractères dans un fichier texte :

Application Java





FLUX

□ Le cycle d'utilisation de lecture ou écriture d'un flux de données est le suivant :

1. Ouverture du flux
2. Tant qu'il y a des données à lire (ou à écrire), on lit (ou on écrit) la donnée suivante dans le flux
3. Fermeture du flux



CLASSE FILE

- ❑ La gestion de fichiers se fait par l'intermédiaire de la classe *java.io.File*
- ❑ Cette classe possède des méthodes qui permettent d'interroger ou d'agir sur le système de fichiers du système d'exploitation
- ❑ Il n'existe pas des classes pour traiter les répertoires car ils sont considérés comme des fichiers
- ❑ Une instance de la classe *java.io.File* est une **représentation logique** d'un **fichier** ou d'un **répertoire** qui peut ne pas exister physiquement sur le disque





CLASSE FILE

□ Un objet de la classe `java.io.File` peut représenter le nom et le chemin d'un fichier ou d'un répertoire sur le disque
`/E/COURS/JAVA/Jeu.txt`

➤ Créer un objet représentant un fichier existant
`File f = new File("Monprog.txt");` (Représente le fichier sur le disque mais ne représente pas son contenu)

➤ Créer un répertoire `File rep = new File("JAVA");`
`rep.mkdir();`



➤ Obtenir le chemin absolu d'un fichier ou d'un répertoire
`System.out.println(rep.getAbsolutePath());`

➤ ...



CLASSE FILE

Liste des principales méthodes de la classe *java.io.File*

Méthode	Rôle
boolean canRead()	indique si le fichier peut être lu
boolean canWrite()	indique si le fichier peut être modifié
boolean createNewFile()	 création d'un nouveau fichier vide
File createTempFile(String, String)	 création d'un nouveau fichier dans le répertoire par défaut des fichiers temporaires. Les deux arguments sont le préfixe et le suffixe du fichier.
File createTempFile(String, String, File)	création d'un nouveau fichier temporaire. Les trois arguments sont le préfixe et le suffixe du fichier et le répertoire.
boolean delete()	détruire le fichier ou le répertoire. Le booléen indique le succès de l'opération



CLASSE FILE

Liste des principales méthodes de la classe *java.io.File*

boolean exists()	indique si le fichier existe physiquement
String getAbsolutePath()	renvoie le chemin absolu du fichier
String getPath	renvoie le chemin du fichier
boolean isAbsolute()	indique si le chemin est absolu
boolean isDirectory()	indique si le fichier est un répertoire
boolean isFile()	indique si l'objet représente un fichier
long length()	renvoie la longueur du fichier
String[] list()	renvoie la liste des fichiers et répertoire contenu dans le répertoire
boolean mkdir()	création du répertoire
boolean mkdirs()	création du répertoire avec création des répertoire manquant dans l'arborescence du chemin
boolean renameTo()	renommer le fichier



CLASSE FILE

Exemple 1 : On va créer un fichier, avec l'extension .txt, et on va l'enregistrer à la racine d'un projet ! : **un fichier test.txt dont voici le contenu :**

Voici une ligne de test.

Voici une autre ligne de test.

Et comme je suis motivé, en voici une troisième !

```
import java.io.File; //Package à importer afin d'utiliser l'objet File
public class Main {
    public static void main(String[] args) {
        File f = new File("test.txt"); //Création de l'objet File
        System.out.println("Chemin absolu du fichier : " + f.getAbsolutePath());
        System.out.println("Nom du fichier : " + f.getName());
        System.out.println("Est-ce qu'il existe ? " + f.exists());
        System.out.println("Est-ce un répertoire ? " + f.isDirectory());
        System.out.println("Est-ce un fichier ? " + f.isFile()); }
}
```



CLASSE FILE

Chemin absolu du fichier : C:\Users\lenovo\eclipse-workspace\coursPoo
\test.txt

Nom du fichier : test.txt

Est-ce qu'il existe ? true

Est-ce un répertoire ? false

Est-ce un fichier ? true



CLASSE FILE

Exemple 2 :

```
import java.io.* ;
public class ExempleFile {
    static public void main (String args []) { new ExempleFile() ; }
    public ExempleFile () { liste (new File ("c:\\")) ; }
    private void liste (File dir) {
        int i;
        if (dir.isDirectory () == true) {
            String fichiers [] = dir.list () ;
            for (i = 0 ; i != fichiers.length ; i++) {
                System.out.print(fichiers [i] + "\\t") ;
                if ((i+1)%4==0) System.out.println(); }
        } else System.err.println (dir + " n'est pas un repertoire") ; }
    }
```



LES FLUX PROPOSÉS PAR JAVA

- Java met toute une panoplie d'objets afin de pouvoir communiquer (rangés dans un package `java.io`)
- Java distingue dans le package `java.io` deux grandes catégories de flux :
 - La famille des **flux de caractères -textes-** (caractères 16 bits) : des données de type caractères
 - La famille des **flux d'octets -binaires-** (information binaires sur 8 bits) : lire ou écrire des données binaires (sons, images,...)



LES FLUX PROPOSÉS PAR JAVA

◆ Classes de base abstraites des flux

	Flux d'octets	Flux de caractères
Flux d'entrée	<i>java.io.InputStream</i>	<i>java.io.Reader</i>
Flux de sortie	<i>java.io.OutputStream</i>	<i>java.io.Writer</i>

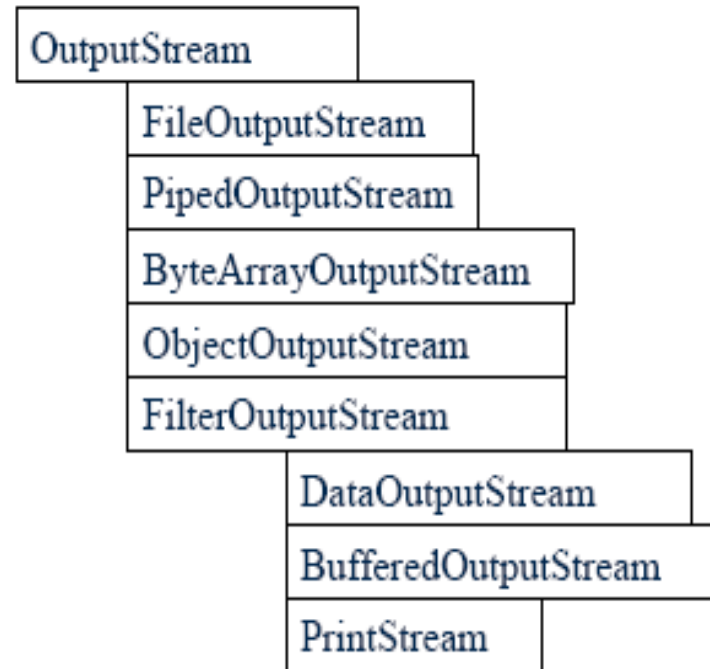
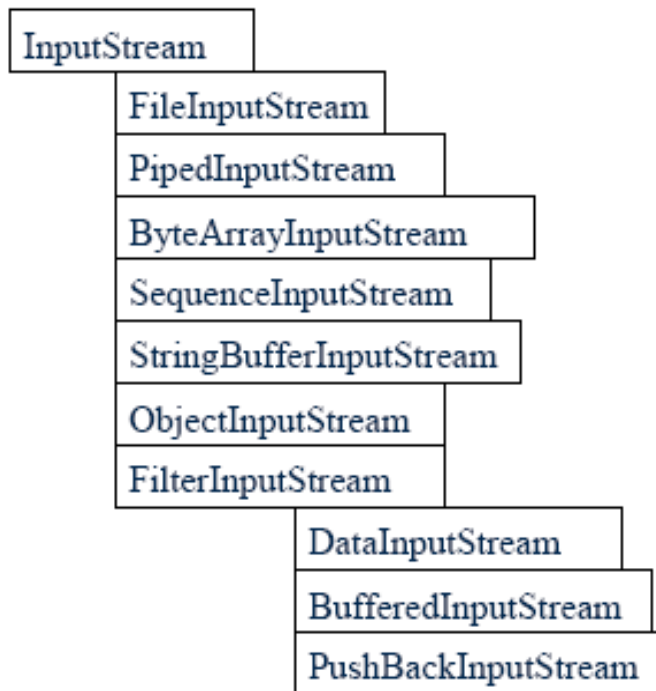
◆ L'API d'E/S JAVA comprend des flux

- de **communication** qui représentent des destinations et des sources (fichiers, socket réseau,...)
- de **traitement** ne fonctionne que s'ils sont chaînés à d'autres flux



FLUX D'OCTETS

- ❑ Ils transportent des données sous forme d'octets. Les flux de ce type sont capables de traiter toutes les données
- ❑ Les classes qui gèrent les flux d'octets héritent d'une des deux classes abstraites **InputStream** ou **OutputStream**.
- ❑ Il existe de nombreuses sous classes pour traiter les flux d'octets





LE FLUX D'OCTETS

□ Fonction des classes de flux d'octets en entrée

Classes utilisées pour la communication

FileInputStream	lecture de fichiers octets par octets.
PipedInputStream	recupère des données provenant d'un flux de sortie (cf. PipedOutputStream).
ByteArrayInputStream	lit des données dans un tampon structuré sous forme d'un array.

Classes utilisées pour le traitement

SequenceInputStream	concaténation d'une énumération de plusieurs flux d'entrée en un seul flux d'entrée.
StringBufferInputStream	lecture d'une String (Sun déconseille son utilisation et préconise son remplacement par <code>StringReader</code>).
ObjectInputStream	lecture d'objets Java.
FilterInputStream	lit à partir d'un InputStream quelconque des données, en "filtrant" certaines données.



FLUX D'OCTETS

□ Fonction des classes de flux d'octets en sortie

Classes utilisées pour la communication

FileOutputStream	écriture de fichiers octets par octets.
PipedOutputStream	envoi des données vers un flux d'entrée (cf. PipedInputStream).
ByteArrayOutputStream	écrit des données dans un tampon structuré sous forme d'un array.

Classes utilisées pour le traitement

ObjectOutputStream	écriture d'objets Java lisibles avec ObjectInputStream.
FilterOutputStream	écrit à partir d'un OutputStream quelconque des données, en "filtrant" certaines données.



FLUX D'OCTETS

□ Les flux d'octets avec un fichier

➤ Les classes **FileInputStream** et **FileOutputStream** permettent de gérer des flux d'octets avec des fichiers

➤ Les flux d'octets en lecture sur un fichier

✓ instancier un objet de la classe **FileInputStream** à partir d'un nom d'un fichier (un fichier)

```
FileInputStream fichier = new FileInputStream("monfichier.dat")
```

//associe l'objet *fichier* à un fichier de nom *monfichier.dat*

✓ **FileInputStream** possède plusieurs constructeurs qui peuvent tous lever l'exception **FileNotFoundException**

Constructeur	Rôle
<code>FileInputStream(String)</code>	Ouvre un flux en lecture sur le fichier dont le nom est donné en paramètre
<code>FileInputStream(File)</code>	Idem mais le fichier est précisé avec un objet de type File



FLUX D'OCTETS

□ Les flux d'octets avec un fichier

➤ Les flux d'octets en lecture sur un fichier

- ✓ plusieurs méthodes de la classe **FileInputStream** permettent de lire un ou plusieurs octets dans le flux
- ✓ toutes ces méthodes peuvent lever l'exception **IOException**
- ✓ **int read()** : méthode envoie la valeur de l'octet lu ou -1 si la fin du flux est atteinte

```
int octet = 0; while (octet != -1 ) {octet = fichier.read();}
```

- ✓ **int read(byte[], int, int)** : méthode lit plusieurs octets
 - Elle attend en paramètre :
 - un tableau d'octets qui contiendra les octets lus
 - l'indice du premier éléments du tableau qui recevra le premier octet
 - et le nombre d'octets à lire
 - Elle renvoie le nombre d'octets lus ou -1 si aucun octet n'a été lus



FLUX D'OCTETS

□ Les flux d'octets avec un fichier.

➤ Les flux d'octets en lecture sur un fichier

- ✓ La lecture d'un **caractère Unicode** demande deux appels de **read()**
 - la lecture d'un entier ou d'un réel se fait octet par octet
- ✓ La classe **FileInputStream** possède plusieurs méthodes pour gérer le flux :

Méthode	Rôle
<code>long skip(long)</code>	saute autant d'octets dans le flux que la valeur fournie en paramètre. Elle renvoie le nombre d'octets sautés.
<code>close()</code>	ferme le flux et libère les ressources qui lui étaient associées
<code>int available()</code>	retourne le nombre d'octets qu'il est encore possible de lire dans le flux



FLUX D'OCTETS

□ Les flux d'octets avec un fichier.

➤ Les flux d'octets en écriture sur un fichier

- ✓ instancier un objet de la classe **FileOutputStream**
- ✓ classe possède plusieurs constructeurs

Constructeur	Rôle
<code>FileOutputStream(String)</code>	Si le fichier précisé n'existe pas, il sera créé. Si il existe et qu'il contient des données celles ci seront écrasées.
<code>FileOutputStream(String, boolean)</code>	Le booléen permet de préciser si les données seront ajoutées au fichier (valeur true) ou écraseront les données existantes (valeur false)

```
FileOuputStream fichier = new FileOutputStream("monfichier.dat");
```



FLUX D'OCTETS

□ Les flux d'octets avec un fichier.

➤ Les flux d'octets en écriture sur un fichier

✓ plusieurs méthodes de la classe **FileOutputStream** permettent d'écrire un ou plusieurs octets dans le flux :

- **write(int)** : méthode écrit l'octet en paramètre dans le flux.
- **write(byte[])** : méthode écrit plusieurs octets. Elle attend en paramètre : un tableau d'octets qui contient les octets à écrire (tous les éléments du tableau sont écrits).
- **write(byte[], int, int)** : méthode écrit plusieurs octets. Elle attend en paramètre :
 - un tableau d'octets qui contient les octets à écrire,
 - l'indice du premier éléments du tableau d'octets à écrire
 - et le nombre d'octets à écrire



FLUX D'OCTETS

□ Les flux d'octets avec un fichier

```
import java.io.*;
public class CopieFichier {
    protected String source;
    protected String destination;
    public CopieFichier(String source, String destination) {
        this.source = source; this.destination = destination;
        copie(); }
    public static void main(String args[]) {
        new CopieFichier("source.txt","copie.txt"); }
    private void copie() {
        try { FileInputStream fis = new FileInputStream(source);
            FileOutputStream fos = new FileOutputStream(destination);
            while(fis.available() > 0) fos.write(fis.read());
            fis.close(); fos.close();
        } catch (Exception e) {e.printStackTrace();}
    }
}
```




FLUX D'OCTETS

□ Les flux filtrés : sont les deux classes abstraites **FilterInputStream** et **FilterOutputStream**

- définissent un comportement global pour ces classes filles qui permettent d'ajouter des fonctionnalités aux flux d'E/S
- 4 classes filles héritant de **FilterInputStream** (idem pour **FilterOutputStream**)
 - **DataInputStream** : offre la possibilité de lire directement des types primitifs (double, char, int), ceci grâce à des méthodes comme **readDouble()** , **readInt()** ...
 - **BufferedInputStream** : permet d'avoir un tampon à disposition dans la lecture du flux
 - **PushbackInputStream** : permet de remettre un octet déjà lu dans le flux entrant
 - **LineNumberInputStream** : offre la possibilité de récupérer le numéro de ligne lue à un instant t



FLUX D'OCTETS

□ Les flux filtrés : les classes **FilterInputStream** et **FilterOutputStream**

➤ 4 classes filles héritant de **FilterInputStream** (idem pour **FilterOutputStream**) :

- Ces classes prennent une instance dérivant des classes **InputStream** (pour les classes héritant de **FilterInputStream**) ou de **OutputStream** (pour les classes héritant de **FilterOutputStream**)
- On peut cumuler les filtres, c.à.d que, vu que ces classes acceptent une instance de leur super-classe, elles acceptent une instance de leur cousin !

```
FileInputStream fis = new FileInputStream(new File("toto.txt"));  
DataInputStream dis = new DataInputStream(fis);  
BufferedInputStream bis = new BufferedInputStream(dis);
```

//Ou en condensé :

```
BufferedInputStream bis = new BufferedInputStream(new  
DataInputStream(FileInputStream(new File("toto.txt"))));
```



FLUX D'OCTETS

▣ Les flux filtrés : les classes `FilterInputStream` et `FilterOutputStream`

➤ Afin de rendre compte des améliorations apportées par ces classes :

- lecture d'un ÉNORME fichier texte (5.65Mo) de façon conventionnelle avec l'objet vu précédemment et avec un buffer!
- Ecrire un code qui permet de tester le temps d'exécution de la lecture :

```
import java.io.BufferedReader;
import java.io.DataInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

public class BufferStream {
    public static void main(String[] args) {

        FileInputStream fis;
        BufferedInputStream bis;
```

```
Temps de lecture avec FileInputStream : 1218
Temps de lecture avec BufferedInputStream : 94
BUILD SUCCESSFUL (total time: 1 second)
```

La différence de temps est ÉNORME ! 1.218 secondes pour la première méthode et 0.094



FLUX D'OCTETS

try {

```
fis = new FileInputStream(new File("teste.txt"));
```

```
bis = new BufferedInputStream(new FileInputStream(new File("teste.txt")));
```

```
byte[] buf = new byte[8];
```

```
long startTime = System.currentTimeMillis(); //On récupère le temps du système
```

```
while(fis.read(buf) != -1); //Inutile de faire des traitements dans notre boucle
```

```
//On affiche le temps d'exécution
```

```
System.out.println("Temps de lecture avec FileInputStream : " +  
                    (System.currentTimeMillis() - startTime));
```

```
startTime = System.currentTimeMillis(); //On réinitialise
```

```
while(bis.read(buf) != -1); //Inutile de faire des traitements dans notre boucle
```

```
//On réaffiche
```

```
System.out.println("Temps de lecture avec BufferedInputStream : " +  
                    (System.currentTimeMillis() - startTime));
```

```
fis.close(); bis.close(); //On ferme nos flux de données
```

```
}
```

```
catch (FileNotFoundException e) { e.printStackTrace(); }
```

```
catch (IOException e) { e.printStackTrace(); }
```

```
}
```

```
}
```



FLUX D'OCTETS

□ Les flux filtrés : les classes `FilterInputStream` et `FilterOutputStream`

- En écriture, la différence de performance devient démentiel (toujours fichier texte (5.65Mo) : 3.546 secondes en temps normal et 0.235 seconde avec un buffer !

```
import java.io.BufferedOutputStream;  
import java.io.BufferedInputStream;  
import java.io.File;  
import java.io.FileInputStream;  
import java.io.FileNotFoundException;  
import java.io.FileOutputStream;  
import java.io.IOException;
```

```
public class BufferStreamEcriture {  
    public static void main(String[] args) {
```

```
        FileInputStream fis;  
        FileOutputStream fos;  
        BufferedInputStream bis;  
        BufferedOutputStream bos;
```

```
Temps de lecture + écriture avec FileInputStream et FileOutputStream : 3546  
Temps de lecture + écriture avec BufferedInputStream et BufferedOutputStream : 235  
BUILD SUCCESSFUL (total time: 3 seconds)
```



FLUX D'OCTETS

```
try {  
    fis = new FileInputStream(new File("teste.txt"));  
    fos = new FileOutputStream(new File("test2.txt"));  
    bis = new BufferedInputStream(new FileInputStream(new File("teste.txt")));  
    bos = new BufferedOutputStream(new FileOutputStream(new File("test3.txt")));  
    byte[] buf = new byte[8];  
  
    long startTime = System.currentTimeMillis(); //On récupère le temps du système  
    while(fis.read(buf) != -1) fos.write(buf);  
    //On affiche le temps d'exécution  
    System.out.println("Temps de lecture + écriture avec FileInputStream et  
                        FileOutputStream : " + (System.currentTimeMillis() - startTime));  
  
    startTime = System.currentTimeMillis(); //On réinitialise  
    while(bis.read(buf) != -1) bos.write(buf);  
    //On réaffiche  
    System.out.println("Temps de lecture + écriture avec BufferedInputStream et  
                        BufferedOutputStream : " + (System.currentTimeMillis() - startTime));  
  
    fis.close(); bis.close(); //On ferme nos flux de données  
}  
catch (FileNotFoundException e) { e.printStackTrace(); }  
catch (IOException e) { e.printStackTrace(); }  
}
```



FLUX D'OCTETS

les objets `Data(InputStream/OutputStream)` peuvent s'utiliser comme des objets `BufferedInputStream`.

Les objets `Data(InputStream/OutputStream)` ont des méthodes de lecture pour chaque type primitif

Nous allons donc créer un fichier de toute pièce pour le lire par la suite.

```
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
```

```
public class DataBufferStream {
    public static void main(String[] args) {
        DataInputStream dis;
        DataOutputStream dos;
```

```
true
100
C
12.05
100.52
1024
123456789654321
2
BUILD SUCCESSFUL (total time: 0 seconds)
```





FLUX D'OCTETS

try {

```
dos = new DataOutputStream( new BufferedOutputStream(  
    new FileOutputStream( new File("sdz.txt"))));
```

```
dos.writeBoolean(true); //Nous allons écrire chaque primitif  
dos.writeByte(100); dos.writeChar('C'); dos.writeDouble(12.05);  
dos.writeFloat(100.52f); dos.writeInt(1024);  
dos.writeLong(123456789654321L); dos.writeShort(2);  
dos.close();
```

//On récupère maintenant les données !

```
dis = new DataInputStream( new BufferedInputStream(  
    new FileInputStream( new File("sdz.txt"))));
```

```
System.out.println(dis.readBoolean()); System.out.println(dis.readByte());  
System.out.println(dis.readChar()); System.out.println(dis.readDouble());  
System.out.println(dis.readFloat()); System.out.println(dis.readInt());  
System.out.println(dis.readLong()); System.out.println(dis.readShort());
```

```
} catch (FileNotFoundException e) {e.printStackTrace(); }
```

```
    catch (IOException e) { e.printStackTrace(); }
```

```
}
```

```
}
```




FLUX D'OCTETS

Les fichiers à accès direct

- Les **fichiers à accès direct** permettent un **accès rapide** à un enregistrement contenu dans un fichier
- Il est possible dans un tel type de fichier de **mettre à jour directement** un de ces enregistrements

Exemple : Le plus simple pour utiliser un tel type de fichier est qu'il contient des enregistrements de taille fixe (mais ce n'est pas obligatoire)





FLUX D'OCTETS

Les fichiers à accès direct

□ La classe `RandomAccessFile` encapsule les opérations de lecture/écriture d'un tel fichier

- implémente les interfaces `DataInput` et `DataOutput`
- possède deux constructeurs qui attendent en paramètre le fichier à utiliser (nom de fichier ou objet de type `File`) et le mode d'accès (`r` : lecture seule ou `rw` : lecture/écriture)
 - Ces deux constructeurs peuvent lever les exceptions suivantes:
 - `FileNotFoundException` si le fichier n'est pas trouvé
 - `IllegalArgumentException` si le mode n'est pas `r` ou `rw`
 - `SecurityException` si le gestionnaire de sécurité empêche l'accès aux fichiers dans le mode précisé



FLUX D'OCTETS

Les fichiers à accès direct

□ La classe `RandomAccessFile` possède de nombreuses méthodes `writeXXX()` pour écrire des types primitifs dans le fichier

Exemple :

```
import java.io.RandomAccessFile;
public class TestRandomAccesFile {
    public static void main(String[] args) {
        try {
            RandomAccessFile monFichier = new RandomAccessFile("monfichier.dat",
"rw");
            for (int i = 0; i < 10; i++) { monFichier.writeInt(i * 100); }
            monFichier.close();
        } catch (Exception e) { e.printStackTrace(); }
    }
}
```



FLUX D'OCTETS

Les fichiers à accès direct

□ La classe `RandomAccessFile` possède aussi de nombreuses classes `readXXX()` pour lire des données primitives dans le fichier

Exemple :

```
import java.io.RandomAccessFile;
public class TestRandomAccesFile {
    public static void main(String[] args)
    try {
        RandomAccessFile monFichier = new RandomAccessFile("monfichier.dat",
"rw");
        for (int i = 0; i < 10; i++) { System.out.println(monFichier.readInt()); }
        monFichier.close();
    } catch (Exception e) { e.printStackTrace();}
}
```

Résultat :

```
0
100
200
300
400
500
600
700
800
900
```





FLUX D'OCTETS

Les fichiers à accès direct

- ❑ Pour naviguer dans le fichier, la classe `RandomAccessFile` utilise un pointeur qui indique la position dans le fichier
 - les opérations de lecture ou de mise à jour doivent être effectuées
- ❑ La méthode `getFilePointer()` permet de connaître la position de ce pointeur et la méthode `seek()` permet de le déplacer
- ❑ La méthode `seek()` attendant en paramètre un **entier long** qui représente la **position**, dans le fichier, précisée en octets (la **première position** commence à **zéro**)



FLUX D'OCTETS

Les fichiers à accès direct

Exemple : lecture de la sixième données

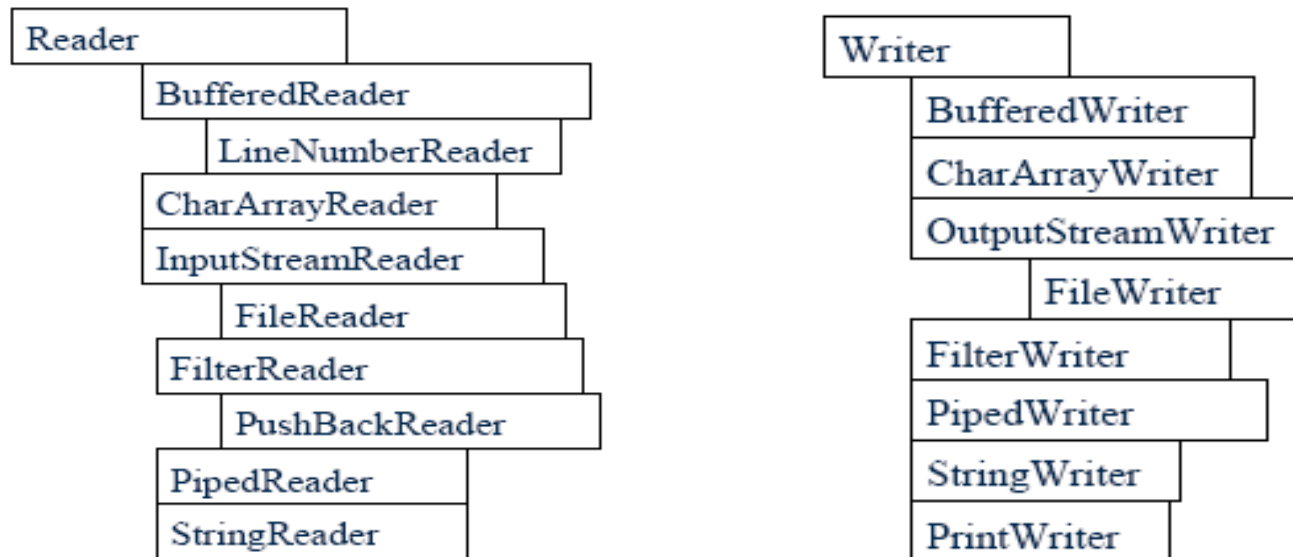
```
import java.io.RandomAccessFile;
public class TestRandomAccesFile {
    public static void main(String[] args) {
        try {
            RandomAccessFile monFichier = new RandomAccessFile("monfichier.dat",
"rw");

                // 5 représente le sixième enregistrement puisque le premier
commence à 0
                // 4 est la taille des données puisqu'elles sont des entiers de type int
                // (codé sur 4 octets)
            monFichier.seek(5*4);
            System.out.println(monFichier.readInt());
            monFichier.close();
        } catch (Exception e) { e.printStackTrace();}
    }
}
```



FLUX DE CARACTÈRES

- ❑ Ils transportent des données sous forme de caractères : java les gère avec le format **Unicode** qui code les caractères sur **2 octets**.
- ❑ Les classes qui gèrent les flux de caractères héritent d'une des deux classes abstraites **Reader** ou **Writer**.
- ❑ Il existe de nombreuses sous classes pour traiter les flux de caractères.





LE FLUX DE CARACTÈRES

□ Fonction des classes de flux de caractères en entrée

BufferedReader	lecture de caractères dans un tampon.
CharArrayReader	lit de caractères dans un tampon structuré sous forme d'un array.
FileReader	lecture de caractères dans un fichier texte.
FilterReader	lit à partir d'un Reader quelconque des caractères, en "filtrant" certaines caractères.
InputStreamReader	conversion de flux d'octets en flux de caractères (8 bits en 16 bits)
LineNumberReader	lecture de caractères dans un tampon (dérive de <code>BufferedReader</code>) avec comptage de lignes.
PipedReader	récupère des données provenant d'un flux de caractères en sortie (cf. <code>PipedWriter</code>).
StringReader	lecture de caractères à partir d'une chaîne de type <code>String</code> .



FLUX DE CARACTÈRES

❑ Fonction des classes de flux de caractères en sortie

BufferedWriter	écriture de caractères dans un tampon.
CharArrayWriter	écrit des caractères dans un tampon structuré sous forme d'un array.
FileWriter	écriture de caractères dans un fichier texte.
FilterWriter	écrit à partir d'un Reader quelconque des caractères, en "filtrant" certaines caractères.
OutputStreamWriter	conversion de flux d'octets en flux de caractères (8 bits en 16 bits)
PipedWriter	envoi des données vers un flux d'entrée (cf. PipedReader).
StringWriter	écriture de caractères dans une chaîne de type String.



FLUX DE CARACTÈRES

□ Classe Reader

- est une classe abstraite qui est la classe mère de toutes les classes qui gèrent des flux de caractères en lecture.
- définit plusieurs méthodes :

Méthodes	Rôles
<code>boolean ready()</code>	indique si le flux est prêt à être lu
<code>close()</code>	ferme le flux et libère les ressources qui lui étaient associées
<code>int read()</code>	renvoie le caractère lu ou <code>-1</code> si la fin du flux est atteinte.
<code>int read(char[])</code>	lire plusieurs caractères et les mettre dans un tableau de caractères
<code>int read(char[], int, int)</code>	lire plusieurs caractères. Elle attend en paramètre : un tableau de caractères qui contiendra les caractères lus, l'indice du premier éléments du tableau qui recevra le premier caractère et le nombre de caractères à lire. Elle renvoie le nombre de caractères lus ou <code>-1</code> si aucun caractère n'a été lu. La tableau de caractères contient les caractères lus.
<code>long skip(long)</code>	saute autant de caractères dans le flux que la valeur fournie en paramètre. Elle renvoie le nombre de caractères sautés.



FLUX DE CARACTÈRES

□ Classe Writer

- est une classe abstraite qui est la classe mère de toutes les classes qui gèrent des flux de caractères en écriture
- définit plusieurs méthodes :

Méthodes	Rôles
<code>close()</code>	ferme le flux et libère les ressources qui lui étaient associées
<code>write(int)</code>	écrire le caractère en paramètre dans le flux.
<code>write(char[])</code>	écrire le tableau de caractères en paramètre dans le flux.
<code>write(char[], int, int)</code>	écrire plusieurs caractères. Elle attend en paramètres : un tableau de caractères, l'indice du premier caractère dans le tableau à écrire et le nombre de caractères à écrire.
<code>write(String)</code>	écrire la chaîne de caractères en paramètre dans le flux
<code>write(String, int, int)</code>	écrire une portion d'une chaîne de caractères. Elle attend en paramètre : une chaîne de caractères, l'indice du premier caractère dans la chaîne à écrire et le nombre de caractères à écrire.



FLUX DE CARACTÈRES

- Les classes **FileReader** et **FileWriter** permettent de gérer des flux de caractères avec des fichiers
- Les flux de caractères en lecture sur un fichier
 - instancier un objet de la classe **FileReader** à partir d'un nom d'un fichier (hérite de la classe **InputStreamReader**)
 - classe possède plusieurs constructeurs qui peuvent tous lever une exception de type **FileNotFoundException**

Constructeur	Rôle
<code>FileReader(String)</code>	Créer un flux en lecture vers le fichier dont le nom est précisé en paramètre
<code>FileReader(File)</code>	mais le fichier est précisé avec un objet de type <code>File</code>

```
FileReader fichier = new FileReader("monfichier.txt");
```



FLUX DE CARACTÈRES

□ Les flux de caractères en écriture sur un fichier

- instancier un objet de la classe **FileWriter** à partir d'un nom d'un fichier (hérite de la classe **OutputStreamWriter**)

```
FileWriter fichier = new FileWriter("monfichier.txt");
```

- classe possède plusieurs constructeurs

Constructeur	Rôle
<code>FileWriter(String)</code>	Si le nom du fichier précisé n'existe pas alors le fichier sera créé. Si il existe et qu'il contient des données celles ci seront écrasées.
<code>FileWriter(File)</code>	Idem mais le fichier est précisé avec un objet de la classe <code>File</code> .
<code>FileWriter(String, boolean)</code>	Le booléen permet de préciser si les données seront ajoutées au fichier (valeur <code>true</code>) ou écraseront les données existantes (valeur <code>false</code>)



FLUX DE CARACTÈRES

□ Les flux de caractères sur un fichier

//Package à importer afin d'utiliser l'objet File

```
import java.io.File;
```

```
import java.io.FileNotFoundException;
```

```
import java.io.FileReader;
```

```
import java.io.FileWriter;
```

```
import java.io.IOException;
```

```
public class Main {
```

```
    public static void main(String[] args) {
```

```
        File file = new File("testFileWriter.txt");
```

```
        FileWriter fw;
```

```
        FileReader fr;
```

```
        try {
```

```
            fw = new FileWriter(file); //Création de l'objet
```

```
            String str = "Bonjour à tous amis 2DSI Al Kendi !\n";
```

```
            str += "\tComment allez-vous ? \n";
```

```
            fw.write(str); //On écrit la chaîne
```

```
            fw.close(); //On ferme le flux
```



FLUX DE CARACTÈRES

□ Les flux de caractères sur un fichier

```
fr = new FileReader(file); //création de l'objet de lecture
str = "";
int i = 0;
while((i = fr.read()) != -1) //Lecture des données
    str += (char)i;
System.out.println(str); //affichage
} catch (FileNotFoundException e) { e.printStackTrace(); }
    catch (IOException e) { e.printStackTrace(); }
}
}
```

la lecture d'un fichier inexistant entraîne une **FileNotFoundException** ,
et l'écriture peut entraîner un **IOException** !



FLUX DE CARACTÈRES

□ Les flux de caractères tamponnés avec un fichier

➤ Les classes **BufferedReader** et **BufferedWriter** permettent de gérer des flux de caractères tamponnés avec des fichiers

- pour améliorer les performances des flux sur un fichier,
- la mise en tampon des données lues ou écrites permet de traiter un ensemble de caractères représentant une ligne plutôt que de traiter les données caractères par caractères

➤ Les flux de caractères tamponnés en lecture avec un fichier

- instancier un objet de la classe **BufferedReader**

BufferedReader fichier = new BufferedReader(new FileReader ("monfichier.txt"));

- possède plusieurs constructeurs qui peuvent tous lever une exception de type **FileNotFoundException**

Constructeur	Rôle
BufferedReader(Reader)	le paramètre fourni doit correspondre au flux à lire.
BufferedReader(Reader, int)	l'entier en paramètre permet de préciser la taille du buffer. Il doit être positif sinon une exception de type IllegalArgumentException est levée.



LE FLUX DE CARACTÈRES

□ Les flux de caractères tamponnés avec un fichier

➤ Les flux de caractères tamponnés en lecture avec un fichier

- plusieurs méthodes de la classe **BufferedReader** héritées de la classe **Reader** permettent de lire un ou + caractères dans le flux
- toutes ces méthodes peuvent lever une exception de type **IOException**
- **BufferedReader** définit une méthode supplémentaire pour la lecture

Méthode	Rôle
<code>String readLine()</code>	lire une ligne de caractères dans le flux. Une ligne est une suite de caractères qui se termine par un retour chariot '\r' ou un saut de ligne '\n' ou les deux.

- La classe **BufferedReader** possède plusieurs méthodes pour gérer le flux héritées de la classe **Reader**
- **close()** : pour libérer les ressources qui sont allouées aux flux



FLUX DE CARACTÈRES

```
import java.io.*;
public class TestBufferedReader {
    protected String source;
    public TestBufferedReader(String source) {
        this.source = source;
        lecture();
    }
    public static void main(String args[]) {
        new TestBufferedReader("source.txt");
    }
    private void lecture() {
        try {
            String ligne ;
            BufferedReader fichier = new BufferedReader(new FileReader(source));
            while ((ligne = fichier.readLine()) != null) System.out.println(ligne);
            fichier.close();
        } catch (Exception e) {e.printStackTrace();}
    }
}
```



FLUX DE CARACTÈRES

□ Les flux de caractères tamponnés avec un fichier

➤ Les flux de caractères tamponnés en écriture avec un fichier

- instancier un objet de la classe **BufferedWriter**
- possède plusieurs constructeurs qui peuvent tous lever une exception de type **FileNotFoundException**

Constructeur	Rôle
BufferedWriter(Writer)	le paramètre fourni doit correspondre au flux dans lequel les données sont écrites.
BufferedWriter(Writer, int)	l'entier en paramètre permet de préciser la taille du buffer. Il doit être positif sinon une exception IllegalArgumentException est levée.

BufferedWriter fichier = new BufferedWriter(new FileWriter("monfichier.txt"));

- + méthodes de la classe **BufferedWriter** héritées de la classe **Writer** permettent de lire un ou + caractères dans le flux
- La classe **BufferedWriter** possède + méthodes pour gérer le flux

Méthode	Rôle
flush()	vide le tampon en écrivant les données dans le flux.
newLine()	écrire un séparateur de ligne dans le flux



FLUX DE CARACTÈRES

```
import java.io.*;
import java.util.*;

public class TestBufferedWriter {
    protected String destination;
    public TestBufferedWriter(String destination) {
        this.destination = destination;
        traitement(); }
    public static void main(String args[]) {
        new TestBufferedWriter("print.txt"); }

    private void traitement() {
        try {
            String ligne ;
            int nombre = 123;
            BufferedWriter fichier = new BufferedWriter( new FileWriter(destination));
            fichier.write("bonjour tout le monde");
            fichier.newLine();
            fichier.write("Nous sommes le " + new Date());
            fichier.write(", le nombre magique est " + nombre);
            fichier.close();
        } catch (Exception e) { e.printStackTrace();}
    }
}
```



EXERCICE 1

La sérialisation : les objets `ObjectInputStream` et `ObjectOutputStream`

`ObjectOutputStream` : permet de sauvegarder l'état d'un objet dans un fichier ou autre. Pour cela, l'objet doit implémenter l'interface *serializable*. Seuls les membres *non-transcients* et *non statiques* seront sauvegardés.

`ObjectInputStream` : permet de restaurer l'état d'un objet (*serializable*) sauvegardé dans un fichier ou autre.

N.B. Seuls les membres non-transcients et non statiques seront sauvegardés.

Soit l'objet à sauvegarder :

```
import java.io.*;

public class Personne implements Serializable{
    private int age;
    private String nom;
    public Personne(int age, String nom) {
        this.age=age;
        this.nom=nom;
    }
    public String toString() { return "nom: "+nom+" age: "+age;
    }
}
```



EXERCICE 1

La sérialisation : les objets `ObjectInputStream` et `ObjectOutputStream`

```
import java.io.*;

public class Serialisation {
    public static void main(String[] args){
        try{
            Personne per1=new Personne(12, "Amine");
            Personne per2=new Personne(16, "Ahmed");
            Personne per3=new Personne(8, "Sana");

            Personne personne;
            //creation du flux
            ObjectOutputStream out=new ObjectOutputStream(new FileOutputStream("fichierObjet"));
            try { //écriture des objets
                out.writeObject(per1);
                out.writeObject(per2);
                out.writeObject(per3);
                out.flush();
            }
            finally { out.close(); }
```



EXERCICE 1

La sérialisation : les objets `ObjectInputStream` et `ObjectOutputStream`

```
//lecture de l'objet
ObjectInputStream in=new ObjectInputStream(new FileInputStream("fichierObjet"));
try{
    for (int i=0; i<=2; i++) {
        personne=(Personne)in.readObject();
        System.out.println(personne); //affichage
    }
}
finally { in.close(); } //fermer le flux
}
catch(Exception e) { System.out.println(e); }
}
}
```



EXERCICE 2

ByteArrayInputStream: permet de lire des données binaires à partir d'un tableau d'octets.

ByteArrayOutputStream: permet d'écrire les données dans un tampon dont la taille s'adapte en fonction du besoin. On peut récupérer les données écrites avec la méthode *toByteArray()* ou bien *toString()*.

```
import java.io.*;

public class TestByteArrayInputStream {
    public static void main(String[] args) {
        ByteArrayInputStream bis;
        int byteLu;
        byte[] buffer=new byte[10]; ByteArrayInputStream
        for(byte i=0;i<10;i++) buffer[i]=i; //remplissage de buffer
        //creation d'un flux d'entrée ayant pour source un tableau de bytes, ici buffer
        bis=new ByteArrayInputStream(buffer);
        //lecture du flux. A noter que les valeurs retournées sont de //type int et non de type byte
        while((byteLu=bis.read())!=-1) {System.out.println(byteLu);}
    }
}
```




EXERCICE 2

ByteArrayInputStream: permet de lire des données binaires à partir d'un tableau d'octets.

ByteArrayOutputStream: permet d'écrire les données dans un tampon dont la taille s'adapte en fonction du besoin. On peut récupérer les données écrites avec la méthode *toByteArray()* ou bien *toString()*.

```
import java.io.*;

public class TestByteArrayOutputStream {
    public static void main(String[] args) {
        ByteArrayOutputStream bos=new ByteArrayOutputStream();
        int byteLu;
        byte[] tab=new byte[10];

        for(byte i=0;i<10;i++) bos.write(i); //écriture dans le tampon
        tab=bos.toByteArray(); //récupération des données contenues dans le tampon
        for(int j=0;j<10;j++) System.out.println(tab[j]); //affichage des données
    }
}
```