

java.io

par Anis Frikha ([Mon site](#))

Date de publication : 08/05/2006

Dernière mise à jour :

Ce tutorial vise à présenter le package java.io en décrivant les différentes classes qui le composent et en précisant quand et comment les utiliser

I - Préface

II - Introduction

III - java.io

III-A - Les flux binaires:

III-A-1 - Les flux d'entrée:

III-A-1-a - Les flux de communication:

III-A-1-b - Les flux de traitement:

III-A-2 - Les flux de sortie:

III-A-2-a - Les flux de communication:

III-A-2-b - Les flux de traitement:

III-B - Les flux de caractères:

III-B-1 - Les flux d'entrée:

III-B-1-a - Les flux de communication:

III-B-1-b - Les flux de traitement:

III-B-2 - Les flux de sortie:

III-B-2-a - Les flux de communication:

III-B-2-b - Les flux de traitement:

III-C - La classe File:

III-D - Récapitulatif:

IV - Conclusion:

V - Remerciement

VI - Téléchargement:

I - Préface

Ce tutorial vise à présenter les packages java.io en décrivant les différentes classes qui les composent et en précisant **quand** et **comment** les utiliser. Il vous aidera, je l'espère, à y voir plus clair dans cette explosion de classes qu'est java.io.

II - Introduction

Une entrée/sortie en Java consiste à **échanger** des données entre le programme et une autre source qui peut être: la mémoire, un fichier, ou le programme lui-même..... .Pour réaliser cela, Java emploie ce qu'on appelle un **stream** (qui signifie flux) entre la source et la destination des données. Toute opération d'entrée/sortie en Java suit le schéma suivant:

- Ouverture d'un flux;
- Lecture ou écriture des données;
- Fermeture du flux.

Vous l'aurez compris, java.io fournit toutes les classes nécessaires à la **création, lecture, écriture et traitement** des flux.

III - java.io

Le package *java.io* comporte plusieurs sortes de flux qui peuvent être classés suivant plusieurs critères:

- les flux d'entrée et les flux de sortie
- les flux de caractères et les flux binaires (dû au fait que java utilise l'unicode donc les caractères sont codés sur 2 octets et non sur un seul)
- les flux de communication et les flux de traitement
- les flux avec ou sans tampon
- ...

La structure que j'ai adopté pour ce tutorial est de d'abord considérer **les flux binaires** avec les flux de communication et ceux de traitement, puis **les flux de caractères** avec eux aussi les flux de communication et de traitement.

III-A - Les flux binaires:

Les flux binaires servent comme leur nom l'indique à manipuler des octets, c'est-à-dire qu'il y a échange d'octets entre le programme et une autre entité extérieur à ce dernier. Ce genre de flux peut servir à charger en mémoire des images ou bien de les enregistrer sur le disque, ils sont aussi utilisés pour enregistrer des objets (procédé nommé sérialisation) ou les charger (désérialisation).

Les flux binaires dérivent de deux classes:

- *InputStream* pour les flux d'entrée et
- *OutputStream* pour les flux de sortie.

Ces deux classes sont abstraites. On ne les utilisera donc pas directement pour obtenir un flux mais on utilisera une de leurs nombreuses classes filles qu'on détaillera par la suite. On notera la présence d'un certains nombres de méthodes, à savoir les différentes méthodes *write* pour *OutputStream* et les différentes méthodes *read* pour *InputStream*. Il est grand temps maintenant de détailler les différentes classes filles et leurs rôles. Nous aborderons tout d'abord les flux d'entrée puis ceux de sortie.

III-A-1 - Les flux d'entrée:

ces flux sont des sous-classes de *InputStream* et peuvent être classés en deux catégories:

- Les flux de communication, servant essentiellement à créer une liaison entre le programme et une autre entité.
- Les flux de traitement qui, comme leur nom l'indique, servent plutôt à traiter les données échangées.

III-A-1-a - Les flux de communication:

Ils sont composés des classes suivantes:

FileInputStream:

permet de créer un flux avec un fichier présent dans le système de fichiers. Cette classe possède un constructeur prenant en paramètre un objet de type *File* (qu'on abordera ultérieurement) et un autre prenant un *String* comme paramètre, qui représente le chemin vers le fichier.

Voici un bout de code permettant de charger le contenu binaire d'un fichier et de l'afficher sur la sortie standard.

FileInputStream

```
import java.io.*;

public class TestFileInputStream {
    public static void main(String[] args){
        FileInputStream fis;
        int byteLu;
        try{
            //creation d'un flux d'entrée ayant pour source un fichier nommé
            //source, cette instantiation peut lever une exception de type
            //FileNotFoundException
            fis=new FileInputStream("source");
            //la methode read() renvoie un int representant l'octet lu, la valeur (-1)
            //represente la fin du fichier.
            // read peut lever une exception du type IOException
            try{
                while((byteLu=fis.read())!=-1){
                    System.out.println(byteLu);
                }
            }
            //Ne pas oublier de fermer le flux afin de liberer les ressources qu'il
            //utilise
            finally{
                fis.close();
            }
        }
        catch(FileNotFoundException ef){
            System.out.println("fichier introuvable");
        }
        catch(IOException e){
            System.out.println(e+"erreur lors de la lecture du fichier");
        }
    }
}
```

On remarque que la sortie de ce programme sont des chiffres compris entre 0 et 255 et qu'il s'agit donc bien d'octets (bytes).

ByteArrayInputStream:

permet de lire des données binaires à partir d'un tableau d'octets. La source de ce flux est donc le programme lui-même.

Voici un exemple de code illustrant son utilisation:

ByteArrayInputStream

```
import java.io.*;

public class TestByteArrayInputStream {
    public static void main(String[] args){
        ByteArrayInputStream bis;
        int byteLu;
    }
}
```

ByteArrayInputStream

```
byte[] buffer=new byte[10];
//remplissage de buffer
for(byte i=0;i<10;i++) {
    buffer[i]=i;
}
//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);
}
}
```

On remarque, d'après l'affichage obtenu, que les valeurs sont lues séquentiellement en partant de l'élément d'indice 0 .

PipedInputStream:

permet de créer une sorte de tube d'entrée (pipe) dans lequel circuleront des octets. Cette classe possède un constructeur ayant pour paramètre un objet de type *PipedOutputStream*. On peut ainsi connecter les deux tubes, c'est-à-dire que ce qui est écrit dans une extrémité peut être lu depuis l'autre.

Dans la pratique, on les utilise pour faire communiquer deux threads. La présence d'un buffer permet de découpler les performances des opérations de lecture et d'écriture et cela sans limite.

A noter que l'usage de ces deux objets depuis un même thread n'est pas recommandé. Un exemple de code sera fourni quand on abordera la classe *PipedOutputStream*.

III-A-1-b - Les flux de traitement:

Ils viennent se greffer sur les flux de communication afin de réaliser un traitement sur les données lues. Ils se composent des classes *BufferedInputStream*, *DataInputStream*, *PushBackInputStream* qui étendent la classe *FilterInputStream*, mais aussi de *SequenceInputStream* et *ObjectInputStream*.

BufferedInputStream:

cette classe permet la lecture de données à l'aide d'un **tampon**. Lorsqu'elle est instanciée, un tableau d'octets est créé afin de servir de tampon. Ce tableau est redimensionné automatiquement à chaque lecture pour contenir les données provenant du flux d'entrée.

Pour comprendre le grand intérêt de cette classe, exécutez les deux codes suivants et remarquez la différence au niveau des performances (Employez un fichier source de quelques Mo).

Avant:

FileInputStream

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

FileInputStream

```
public class TestFileInputStream2 {
    public static void main(String[] args){
        FileInputStream fis;
        int byteLu;
        try{
            //creation d'un flux d'entrée ayant pour source un fichier nommé
            //source,cette instanciation peut lever une exception de type
            //FileNotFoundException
            fis=new FileInputStream("source");
            //lecture des données
            try{
                long startChrono=System.currentTimeMillis();
                System.out.println("debut lecture");
                while((byteLu=fis.read())!=-1);
                System.out.println("fin lecture");
                System.out.println("durée="+ (System.currentTimeMillis()-startChrono));
            }
            finally{
                //fermeture du flux
                fis.close();
            }
        }
        catch(FileNotFoundException ef){
            System.out.println("fichier introuvable");
        }
        catch(IOException e){
            System.out.println(e);
        }
    }
}
```

Après:

BufferedInputStream

```
import java.io.*;

public class TestBufferedInputStream {
    public static void main(String[] args){
        BufferedInputStream bis;
        int byteLu;
        Byte[] buffer;
        try{
            //creation d'un BufferedInputStream sur un InputStream ayant pour
            //source un fichier nommé source,
            bis=new BufferedInputStream(new FileInputStream("source"));
            //la methode available permet de connaitre le nombre de bytes qui
            //pourront être lus d'une manière non bloquante.
            System.out.println(bis.available());
            //lecture des données
            System.out.println("debut lecture");
            try{
                long startChrono=System.currentTimeMillis();
                while((byteLu=bis.read())!=-1);
                System.out.println("fin lecture");
                System.out.println("durée="+ (System.currentTimeMillis()-startChrono));
            }
            finally{
                //ne pas oublier de refermer le flux
                bis.close();
            }
        }
        catch(FileNotFoundException ef){
            System.out.println("fichier introuvable");
        }
    }
}
```

BufferedInputStream

```
catch(IOException e){
    System.out.println(e);
}
}
```

DataInputStream:

sert à lire des données représentant des types primitifs de Java (int, boolean, double, byte, ...) qui ont été préalablement écrits par un *DataOutputStream*.

Ainsi cette classe possède des méthodes comme : *readInt()*, *readBoolean()*, ...

Un exemple de son utilisation sera donné ultérieurement.

SequencelInputStream:

permet de concaténer deux ou plusieurs *InputStream*. La lecture se fait séquentiellement en commençant par le premier et en passant au suivant dès qu'on a atteint la fin du flux courant, tout en appelant sa méthode *close()*.

ObjectInputStream:

permet de «désérialiser» un objet, c'est-à-dire de restaurer un objet préalablement sauvegardé à l'aide d'un *ObjectOutputStream*. La sérialisation sera vue plus en détails lorsqu'on abordera la classe *ObjectOutputStream*.

III-A-2 - Les flux de sortie:

Pour chaque flux d'entrée qu'on a vu, correspond un flux de sortie. Cette partie ne devrait donc pas poser de problème vu que les classes qui y seront présentées font le travail inverse de celles abordées plus haut.

Les flux de sortie sont aussi classés en flux de communication et flux de traitement.

III-A-2-a - Les flux de communication:

Par symétrie ,voici les classes correspondantes:

FileOutputStream:

permet l'écriture séquentielle de données dans un fichier.

Voici un bout de code réalisant la copie d'un fichier.

FileOutputStream

FileOutputStream

```
import java.io.*;

public class CopieFichierSimple {
    public static void main(String[] args){
        FileInputStream fis;
        FileOutputStream fos;
        int byteLu;
        try{
            //creation d'un flux d'entrée ayant pour source un fichier nommé
            //source,cette instanciation peut lever une exception de type
            //FileNotFoundException
            fis=new FileInputStream("source");
            try {
                //création d'un flux de sortie vers un fichier nommé dist, s'il n'existe
                //pas il sera crée sinon il sera écrasé
                fos=new FileOutputStream("dist");
                try {
                    //on lit octet par octet depuis le fichier source et on écrit dans le fichier

                    //dist
                    while((byteLu=fis.read())!=-1) {
                        fos.write(byteLu);
                    }
                }finally{
                    //on ferme fos
                    fos.close();
                }
            }
            finally{
                //on libère les ressources en fermant fis
                fis.close();
            }
        }catch(FileNotFoundException ef){
            System.out.println("fichier introuvable");
        }
        catch(IOException e){
            System.out.println(e);
        }
    }
}
```

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()*.

A noter qu'un appel à la méthode *close()* aussi bien pour cette classe que pour *ByteArrayInputStream* est sans effet.

Voici un exemple d'utilisation de cette classe:

ByteArrayOutputStream

```
import java.io.*;

public class TestByteArrayOutputStream {
    public static void main(String[] args){
        ByteArrayOutputStream bos=new ByteArrayOutputStream();
        int byteLu;
        byte[] tab=new byte[10];
        //écriture dans le tampon
    }
}
```

ByteArrayOutputStream

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

PipedOutputStream:

Permet la création d'un tube (pipe)

Voici l'exemple de code, comme promis :)

première classe :

ThreadEcriturePipeOutputStream

```
import java.io.*;

public class ThreadEcriture implements Runnable{
    private PipedOutputStream pos;

    public ThreadEcriture(PipedOutputStream pos){
        this.pos=pos;
    }

    public void run(){
        try{
            for(int i=0;i<50;i++){
                pos.write(i);
                System.out.println("threadEcriture: j'ai ecrit "+i);
                Thread.sleep(1000);
            }
        }
        catch(Exception e){
            System.out.println(e);
        }finally{
            try{
                pos.close();
            }catch(IOException eio){
                System.out.println(eio);
            }
        }
    }
}
```

deuxième classe :

```
import java.io.*;

public class ThreadLecture implements Runnable{
```

```
private PipedInputStream pis;
int i;

public ThreadLecture(PipedInputStream pis){
    this.pis=pis;
}

public void run(){
    try{
        while(( i=pis.read())!=-1){
            System.out.println("threadLecture: j'ai lu "+i);
        }
    }catch(IOException e){
        System.out.println(e);
    }finally{
        try{
            pis.close();
        }catch(IOException eio){
            System.out.println(eio);
        }
    }
}
}
```

classe principale :

```
import java.io.*;

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

        try{
            //création des deux pipes et leur connexion
            PipedOutputStream pos = new PipedOutputStream();
            PipedInputStream pis = new PipedInputStream(pos);

            ThreadEcriture te = new ThreadEcriture(pos);
            ThreadLecture tl = new ThreadLecture(pis);

            Thread premier = new Thread(te);
            Thread second = new Thread(tl);

            premier.start();
            second.start();

        }
        catch(IOException e){
            System.out.println(e);
        }
    }
}
```

III-A-2-b - Les flux de traitement:

BufferedOutputStream:

permet d'écrire dans un tampon puis d'écrire le tampon en entier sur le fichier au lieu d'écrire octet par octet sur le fichier.

Voici une autre version du code permettant la copie d'un fichier.

BufferedOutputStream

```
import java.io.*;

public class CopieFichierAvecTampon {
    public static void main(String[] args){
        int i;
        try{
            //création des flux
            BufferedInputStream in=new BufferedInputStream(new FileInputStream("source"));
            try{
                BufferedOutputStream out=new BufferedOutputStream(new FileOutputStream("dist"));
                //copie du fichier
                try{
                    while((i=in.read())!=-1) {
                        out.write(i);
                    }
                    //forcer l'écriture du contenu du tampon dans le fichier
                    out.flush();
                }finally{
                    //fermeture de out
                    out.close();
                }
            }
            finally{
                //fermeture de in
                in.close();
            }
        }
        catch(IOException e){
            System.out.println(e);
        }
    }
}
```

On remarque une nette différence de point de vue rapidité de la copie avec la classe CopieFichierSimple.

DataOutputStream:

cette classe permet l'écriture de données sous format Java et assure une portabilité inter-applications et inter-systèmes.

Voici un exemple de code montrant son utilisation:

premier programme réalisant l'écriture:

```
import java.io.*;

public class TestDataOutputStream {
    public static void main(String[] args){
        try{
            //creation du flux
            DataOutputStream out=new DataOutputStream(new FileOutputStream("sortie"));
        }
    }
}
```

```
//données à écrire
boolean test=true;
int i=100;
try{
    //écriture des données
    out.writeBoolean(test);
    out.writeInt(i);

    //vider le buffer
    out.flush();
}
finally{
    //fermer le flux
    out.close();
}
catch(IOException e){
    System.out.println(e);
}
}
```

deuxième programme réalisant la lecture:

```
import java.io.*;

public class TestDataInputStream {
    public static void main(String[] args){
        try{
            //creation du flux
            DataInputStream in=new DataInputStream(new FileInputStream("sortie"));

            //données à lire
            boolean test;
            int i;
            try{
                //lecture des données
                test=in.readBoolean();
                i=in.readInt();
            }
            finally{
                //fermer le flux
                in.close();
            }
            //affichage des données
            System.out.println(test);
            System.out.println(i);
        }
        catch(IOException e){
            System.out.println(e);
        }
    }
}
```

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-transients et non statiques seront sauvegardés.

Voici un exemple illustrant la manière de sérialiser/désérialiser un objet:

l'objet à sauvegarder:

```
import java.io.*;

public class Personne implements Serializable{

    private int age;
    private String nom;

    public Personne(){
        this.age=22;
        this.nom="toto";
    }

    public String toString(){
        return "nom: "+nom+" age: "+age;
    }
}
```

la classe principale :

ObjectOutputStream

```
import java.io.*;

public class Serialisation {
    public static void main(String[] args){
        try{
            Personne per=new Personne();
            Personne personne;
            //creation du flux
            ObjectOutputStream out=new ObjectOutputStream(new FileOutputStream("fichierObjet"));
            try{
                //écriture de l'objet
                out.writeObject(per);
                out.flush();
            }
            finally{
                out.close();
            }
            //lecture de l'objet
            ObjectInputStream in=new ObjectInputStream(new FileInputStream("fichierObjet"));
            try{
                personne=(Personne)in.readObject();
                //affichage
                System.out.println(personne);
            }
            finally{
                //fermer le flux
                in.close();
            }
        }
        catch(Exception e){
            System.out.println(e);
        }
    }
}
```

III-B - Les flux de caractères:

La question qu'on pourrait se poser est: **Pourquoi des flux spéciaux pour les caractères alors que ce sont des données binaires ?**

La réponse est que Java, contrairement à beaucoup d'autres langages, utilise Unicode. Ainsi, les caractères sont codés sur 16 bits et non sur 8. Ces flux servent donc à gérer les jeux de caractères (conversion possible d'un jeu à l'autre). Néanmoins ces classes présentent une certaine analogie avec ce qui a été vu précédemment et ne devraient donc pas poser de problème. Commençons sans plus attendre la présentation de ces classes.

III-B-1 - Les flux d'entrée:

Ils étendent pour la plupart la classe abstraite *Reader* et définissent sa méthode *read*.

III-B-1-a - Les flux de communication:

CharArrayReader:

c'est l'équivalent de *ByteArrayInputStream* pour les caractères. Il utilise aussi un tampon indexé pour la lecture des caractères.

Exemple:

CharArrayReader

```
import java.io.*;

public class TestCharArrayReader {
    public static void main(String[] args){
        try{
            char[] tab={'a','b','c','d','e'};

            //creation du flux
            CharArrayReader car=new CharArrayReader(tab);

            //lecture et affichage des données
            for(int i=0;i < 5;i++){
                System.out.println((char)car.read());
            }

        }catch(IOException e){
            System.out.println(e);
        }
    }
}
```

StringReader:

permet la lecture de caractères à partir d'une chaîne. La chaîne en question est alors la source du flux .

Exemple:

StringReader

```
import java.io.*;

public class TestStringReader {
    public static void main(String[] args){
        try{
            int i;
            String chaine="toto va à l'école";

            //creation du flux
            StringReader sr=new StringReader(chaine);

            //lecture et affichage des données
            while((i=sr.read())!=-1){
                System.out.println((char)i);
            }

        }catch(IOException e){
            System.out.println(e);
        }
    }
}
```

PipedReader:

a un comportement très similaire à *PipedInputStream* : elle se connecte à un *PipedWriter* afin de créer un tube pour l'échange de caractères.

FileReader:

c'est sûrement la classe la plus importante car c'est certainement la plus utilisée. Elle étend *InputStreamReader* et utilise le tampon et l'encodage par défaut. Elle convient donc dans la plupart des cas.

Si on a besoin d'un autre encodage on doit étendre *InputStreamReader*.

Exemple:

FileReader

```
import java.io.*;

public class TestFileReader {
    public static void main(String[] args){
        try{
            int i;
            //creation du flux
            FileReader fr=new FileReader("source");
            try{
                //lecture et affichage des données
                while((i=fr.read())!=-1){
                    System.out.println((char)i);
                }
            }
        }finally{
            fr.close();
        }
    }
}
```

FileReader

```
}  
    catch(IOException e){  
        System.out.println(e);  
    }  
}
```

III-B-1-b - Les flux de traitement:

InputStreamReader:

Cette classe permet de transformer un flux de données binaires en un flux de caractères. Elle est très utile lorsque le tampon ou l'encodage par défaut de *FileReader* ne conviennent pas. Elle possède un constructeur qui prend en paramètres un flux d'entrée de données binaires et un *Charset* (objet servant à définir l'encodage à utiliser). La méthode *read()* lit le nombre nécessaire d'octets constituant un caractère.

BufferedReader:

Cette classe permet l'emploi d'un tampon (dont on peut spécifier la taille) lors de la lecture d'un flux de caractères. Elle est très utile pour améliorer la performance de l'opération de lecture. Son emploi est analogue à celui de *BufferedInputStream*.

LineNumberReader:

sous-classe de *BufferedReader*, elle en hérite donc l'utilisation d'un tampon et permet en plus de lire ligne par ligne (tout en les comptant) grâce à la méthode *readLine()*.

Exemple:

LineNumberReader

```
import java.io.*;  
  
public class TestLineNumberReader {  
    public static void main(String[] args){  
        try{  
            String ligneLue;  
            //creation du flux  
            LineNumberReader lnr=new LineNumberReader(new FileReader("source"));  
            try{  
                //lecture et affichage des données  
                while((ligneLue=lnr.readLine())!=null){  
                    System.out.println(ligneLue);  
                }  
            }  
            finally{  
                //libération des ressources  
                lnr.close();  
            }  
        }  
        catch(IOException e){  
            System.out.println(e);  
        }  
    }  
}
```

LineNumberReader

```
}  
}  
}
```

III-B-2 - Les flux de sortie:

A chaque flux d'entrée correspond un flux de sortie sauf pour la classe *LineNumberReader*. Ils étendent tous la classe *Writer* et redéfinissent, en plus de la méthode *write()*, la méthode *flush()* qui vide le tampon en forçant l'écriture effective des caractères présents dans ce dernier.

III-B-2-a - Les flux de communication:

CharArrayWriter:

permet l'écriture de caractères dans un tampon dont la taille varie afin de contenir les données. A noter que la méthode *close()* n'a aucun effet sur une instance de cette classe et que l'appel d'autres méthodes après *close()* ne lèvent pas d'*IOException*.

StringWriter:

permet l'écriture de caractères dans un *StringBuffer*. Son contenu peut être obtenu sous forme de *String* grâce à la méthode *toString()*.

PipedWriter:

permet de créer un tube où circuleront les caractères en se connectant à un objet de type *PipedReader*.

FileWriter:

Elle convient pour la plupart des cas d'écriture dans un fichier, son fonctionnement est l'opposé de *FileReader*. Cependant elle utilise aussi le tampon et l'encodage par défaut.

III-B-2-b - Les flux de traitement:

OutputStreamWriter:

convertit un flux de données binaires en un flux de caractères.

PrintWriter:

permet d'écrire des caractères formatés, très utile pour l'affichage en mode texte.

BufferedWriter:

permet l'utilisation d'un tampon et donc d'écrire les caractères par bloc.

Exemple:

BufferedWriter

```
import java.io.*;

public class TestBufferedWriter {
    public static void main(String[] args){
        try{
            String ligne;
            //creation des flux
            BufferedReader in=new BufferedReader(new FileReader("source"));
            //lecture et copie des données
            try{
                BufferedWriter out=new BufferedWriter(new FileWriter("dist"));
                try{
                    while((ligne=in.readLine())!=null){
                        out.write(ligne);
                        //insérer un saut de ligne d'une manière portable
                        out.newLine();
                    }
                    out.flush(); //vider le buffer
                }finally{
                    //fermeture de out
                    out.close();
                }
            }finally{
                in.close();
            }
        }catch(IOException e){
            System.out.println(e);
        }
    }
}
```

III-C - La classe File:

La classe *File* est un peu à part dans le package java.io vu qu'elle ne représente pas un flux. Elle sert à encapsuler un fichier ou un répertoire présent dans le système de fichier. Elle permet donc d'obtenir une multitude d'informations comme, par exemple, la taille, si c'est un répertoire ou un fichier,...

Je vous invite à consulter la page de la javadoc la concernant pour avoir une liste exhaustive des méthodes qui peuvent être appliquées.

III-D - Récapitulatif:

Voici un tableau récapitulatif des différentes classes qu'on a vu jusqu'à présent. Néanmoins j'ai opté pour une autre façon de les classer à savoir la destination vu que tous les flux établissent une liaison entre le programme et une destination.

Destination	flux binaires	flux de caractères
La mémoire	ByteArrayInputStream	CharArrayReader
	ByteArrayOutputStream	CharArrayWriter
	StringBufferInputStream	StringReader
		StringWriter
Un fichier	FileInputStream	FileReader
	FileOutputStream	FileWriter
Un chaînage de flux	PipedInputStream	PipedReader
	PipedOutputStream	PipedWriter

IV - Conclusion:

Il en ressort de cet article que java.io est une api très complète avec laquelle il est possible de faire une infinité de choses et conviendra à la plupart des besoins en matière d'entrée/sortie.

Je vous invite tout de même à regarder ces quelques classes qui, certes ne font pas partie de java.io, mais servent néanmoins à manipuler des flux. Je veux parler de: *CheckedInputStream*, *InflaterInputStream* et ses trois sous-classes *GZIPInputStream*, *ZipInputStream*, et *JarInputStream* ainsi que leurs homologues pour les flux de sortie.

Toutes ces classes font partie du package java.util.zip. Il y a aussi la classe *ProgressMonitorInputStream* du package *javax.swing*, ainsi que la classe *DigestOutputStream* du package *java.security*. Cette conclusion aurait été incomplète sans la mention du package *java.nio* et ses sous-packages qui a été créée pour accroître les performances de certaines utilisation de java.io.

J'espère que cet article vous a aidé à mieux comprendre java.io.

N'hésitez surtout pas à me faire des remarques. Je suis ouvert à toute critique (et à tout remerciement aussi :))

V - Remerciement

Je tiens à remercier toute l'équipe responsable de la rubrique java, et tout particulièrement **adiGuba** , **vbrabant** , et **christopheJ** pour leur relecture, leur gentillesse et leur patience.

VI - Téléchargement:

Pour télécharger les sources du tutorial: [ici](#)

