UEF4.3.Programmation Orientée Objet

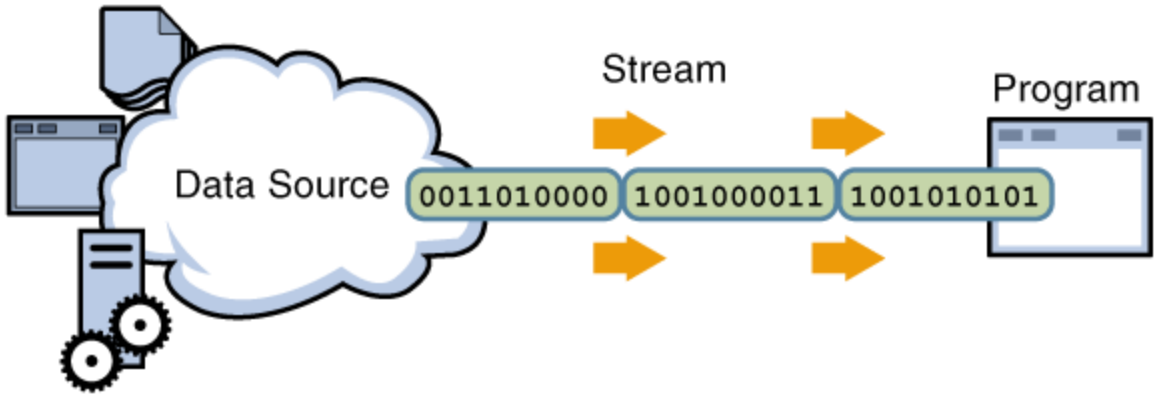
Mme SADEG
s_sadeg@esi.dz
Ecole nationale Supérieure d'Informatique (ESI)
2010/2011

Chapitre VII

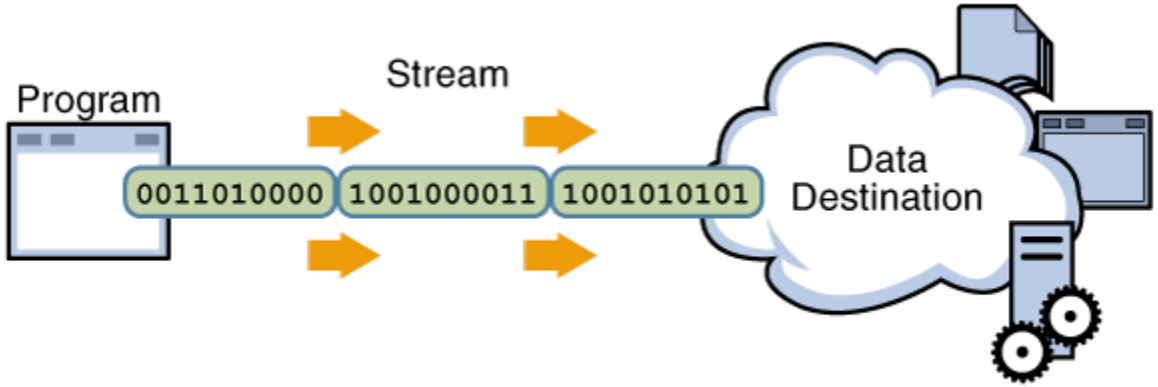
Flux et fichiers

Flux (streams)

- La notion de flux est très générale. Un flux d'entrée (sortie) désigne n'importe quel canal pouvant délivrer(recevoir) de l'information.
- Il peut s'agir d'un fichier, un périphérique, un emplacement mémoire, ou en encore d'une connexion à un site distant.
- Un flux peut supporter plusieurs types de données (binaires, textes)
- Un flux traite toujours les données d'une manière séquentielle



Reading information into a program.



Writing information from a program.

Utilisation d’un flux

- Pour accéder à une information donnée, un programme ouvre un flux vers la source d'informations (un fichier, mémoire etc.) et lie cette information de manière séquentielle.
- De la même manière que la lecture, pour écrire une information donnée, un programme ouvre un flux vers la sortie et la aussi, il écrit de manière séquentielle cette information sur le support d'écriture (fichier, mémoire etc.)

Utilisation d’un flux

Le cycle d’utilisation de lecture ou écriture séquentielle d’un flux de données est le suivant :

1. Ouvrir le flux
2. Tant qu’il y a des données à lire (ou à écrire),

   lire (ou écrire) la donnée suivante dans le flux
3. Fermer le flux

Types de flux

- Les flux données sont divisés en 2 catégories, basées sur le type de données: binaire ou texte.

  - **Flux binaire (octets):** sert à lire/écrire des informations codées sur 8 bits. l'information est transmise sans modification de la mémoire au flux ou du flux à la mémoire. Contient les superclasses: InputStream et OutputStream.

  - **Flux texte (caractères):** sert à lire/écrire des informations textuelles codées sur 16 bits (donc unicode compris). l'information subit une transformation, nommée formatage, de manière que le flux reçoive ou transmette, en définitive, une suite de caractères. Contient les superclasses: Reader et Writer.

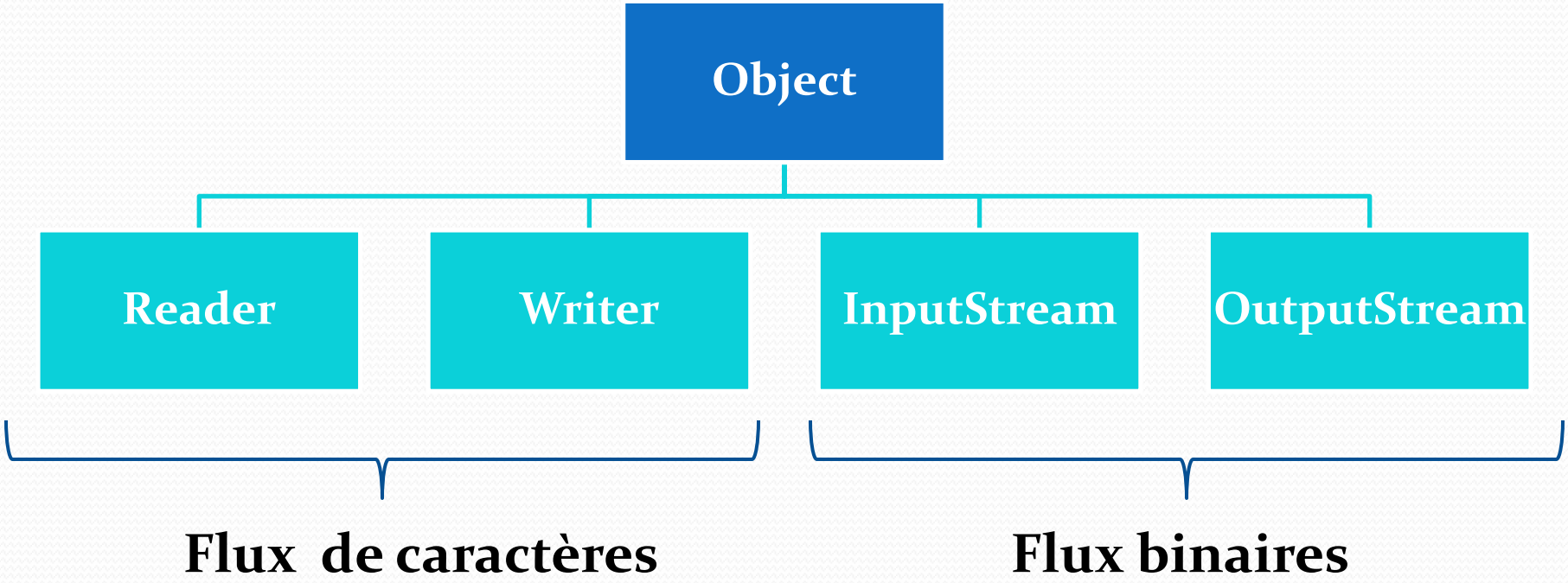
Flux standard

- Trois objets de flux sont crées automatiquement:
  - System.in (lecture)
  - System.out (écriture)
  - System.err (erreur standard, écriture)
- Ces flux se trouvent dans le paquetage: java.lang.System
- À ces flux là, vont s'ajouter une série de classes qui gèrent des flux de données. Ces classes se trouvent dans le paquetage java.io.*

Paquetage java.io

- Il contient la plupart des classes liées aux entrée-sorties (plus de 50)
- Le nom des classes se décompose en un préfixe et 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).

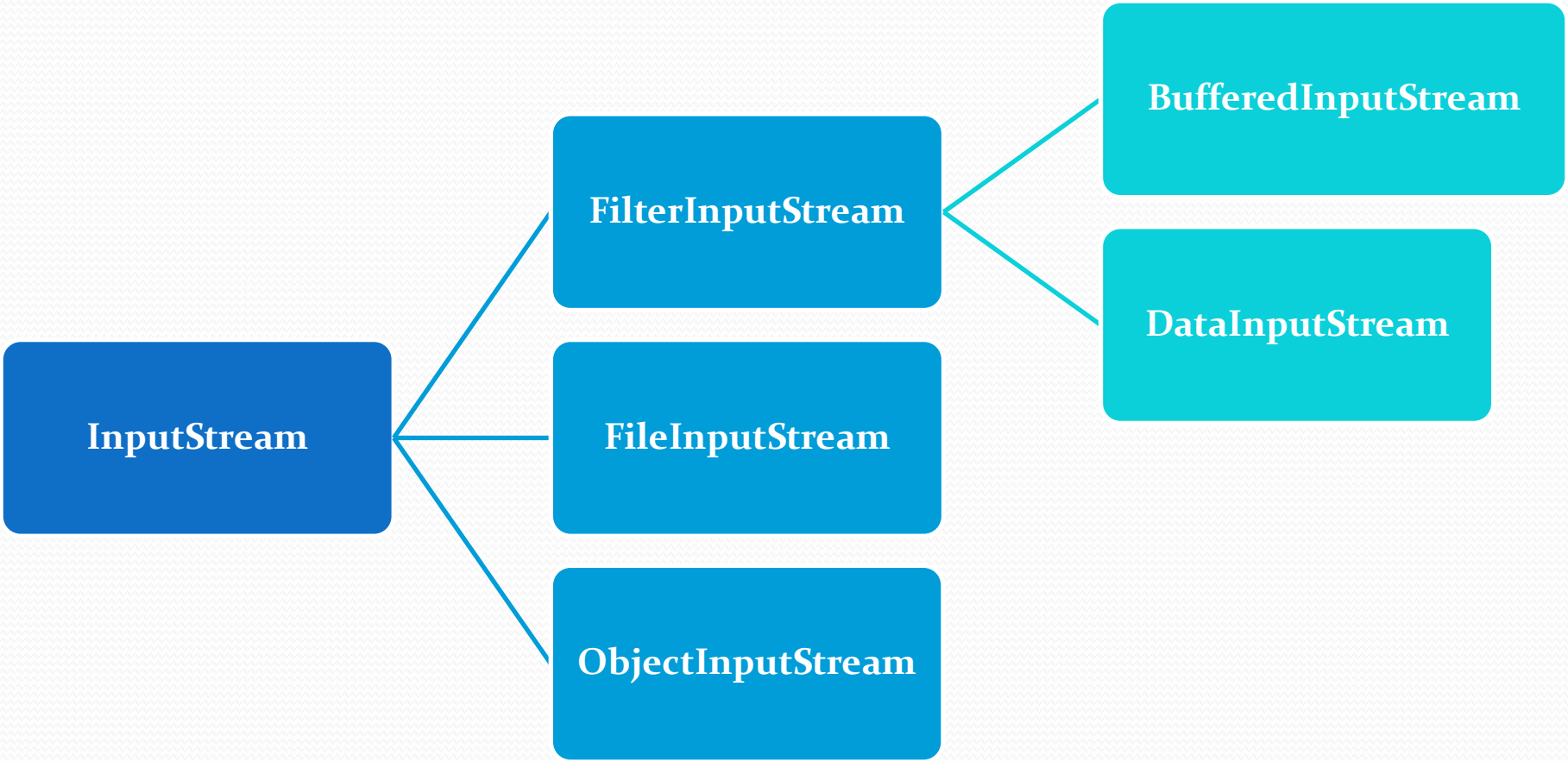
	Flux d'octets	**Flux de caractères**
Flux d'entrée	InputStream	Reader
Flux de sortie	OutputStream	Writer

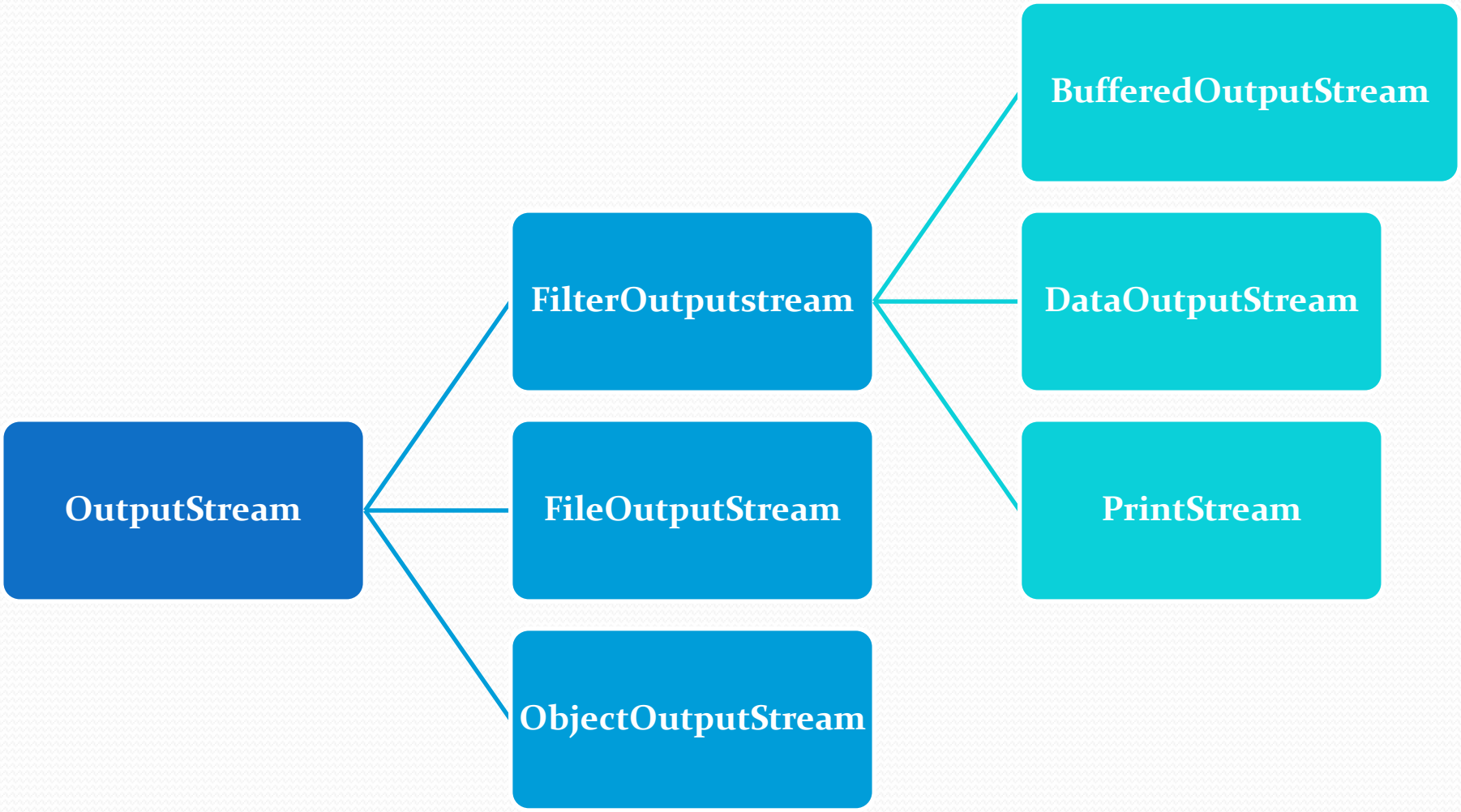

Object
Reader
Writer
InputStream
OutputStream
Flux de caractères
Flux binaires

Flux d’octets (binaires)

Flux d’octets

- Ils transportent des données sous forme d'octets et sont capables de traiter tous les types de 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.


InputStream
FilterInputStream
FileInputStream
ObjectInputStream
BufferedInputStream
DataInputStream


OutputStream
FilterOutputstream
FileOutputStream
ObjectOutputStream
BufferedOutputStream
DataOutputStream
PrintStream

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

- Il faut instancier un objet de la classe FileInputStream. Cette classe possède plusieurs constructeurs qui peuvent tous lever l'exception FileNotFoundException:

```
FileInputStream  fichier = new FileInputStream(“exmplefichier.dat");
```

Constructeur	Rôle
FileInputStream (String)	Ouvre un flux en lecture sur le fichier dont le nom est donné en paramètre
FIleInputStream(File)	Ouvre un flux en lecture sur le fichier donné avec un objet de type File

Qulques méthodes de FileInputStream (1)

- Il existe plusieurs méthodes de la classe FileInputStream qui permettent de lire un ou plusieurs octets dans le flux. Toutes ces méthodes peuvent lever l'exception IOException.
- **int read()**
  - envoie la valeur de l'octet lu ou -1 si la fin du flux est atteinte.
- **Int read(byte[], int, int)**
  - lit plusieurs octets. Elle attend en paramètre : un tableau d'octets qui contiendra les octets lus, l'indice du premier élément 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 lu. Le tableau d'octets contient les octets lus.

Qulques méthodes de FileInputStream (2)

- **long skip(long n)**
  - saute n octets dans le flux
  - renvoie le nombre d'octets réellement sautés
- **int available()**
  - renvoie le nombre d'octets prêts à être lus
- **void close()**
  - ferme le flux.

Il est important de fermer les flux qui ne sont plus utilisés (sauf exceptions signalées dans la javadoc) car des données du flot peuvent être perdues si le flux n'est pas fermé. De plus, la fermeture des flux permet d'économiser des resources

Les flux d'octets en écriture sur un fichier

- Il faut instancier un objet de la classe FileOutputStream. Cette classe possède plusieurs constructeurs :

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

Constructeur	Rôle
FileOutputStream (String)	Si le fichier n'existe pas il sera créé, S'il existe, son contenu sera écrasé
FileoutputStream (String, boolean)	Si le booléen est à true les données seront ajoutées au fichier sinon (false), les données existantes seront écrasées

Quelques méthodes de FileOutPutStream

- Il existe plusieurs méthodes de la classe FileOutputStream qui permettent de lire un ou plusieurs octets dans le flux.

- **write(int)** :
  - écrit l'octet en paramètre dans le flux.
- **write(byte[])**
  - 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)**
  - écrit plusieurs octets. Elle attend en paramètre : un tableau d'octets qui contient les octets à écrire, l'indice du premier élément du tableau d'octets à écrire et le nombre d'octets à écrire.
- **void flush()**
  - Vide le tampon en écrivant les données dans le flux.
- **void close()**

- Là encore, une fois les traitements sur le flux terminés, il faut libérer les ressources qui lui sont allouées en utilisant la méthode close().

```java
import java.io.*;

public class FluxBinaire {
    public static void main(String[] args) throws IOException {
        FileInputStream in = null;
        FileOutputStream out = null;
        try {
            in = new FileInputStream("source.txt");
            out = new FileOutputStream("destination.txt");
            int c;

            while ((c = in.read()) != -1) {
                out.write(c);
            }

        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

Les flux d'octets tamponnés avec un fichier

- Il est possible de doter un flux d'un *tampon*.
- Il s'agit d'un emplacement mémoire qui sert à optimiser les échanges avec le flux en réduisant le nombre d'opérations effectuées.
- Les informations sont d'abord enregistrées dans le tampon, et ce n'est que lorsque ce dernier est plein qu'il est « rempli » à partir du flux ou bien "vidé" dans le flux.
- Pour doter un flux de type *FileIntputStream (FileOutputStream)*d'un tampon, on crée un objet de type BufferedInputStream (BufferedOutputStream) en donnant le premier comme argument au constructeur du second.

```java
public class Bufferedfluxbinaire {
    public static void main(String[] args) throws IOException {
        BufferedInputStream in = null;
        BufferedOutputStream out = null;
        try {
            in = new BufferedInputStream(new FileInputStream("source.txt"));
            out = new BufferedOutputStream(new
                        FileOutputStream("destination.txt"));
            int c;

            while ((c = in.read()) != -1) {
                out.write(c);
            }

        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

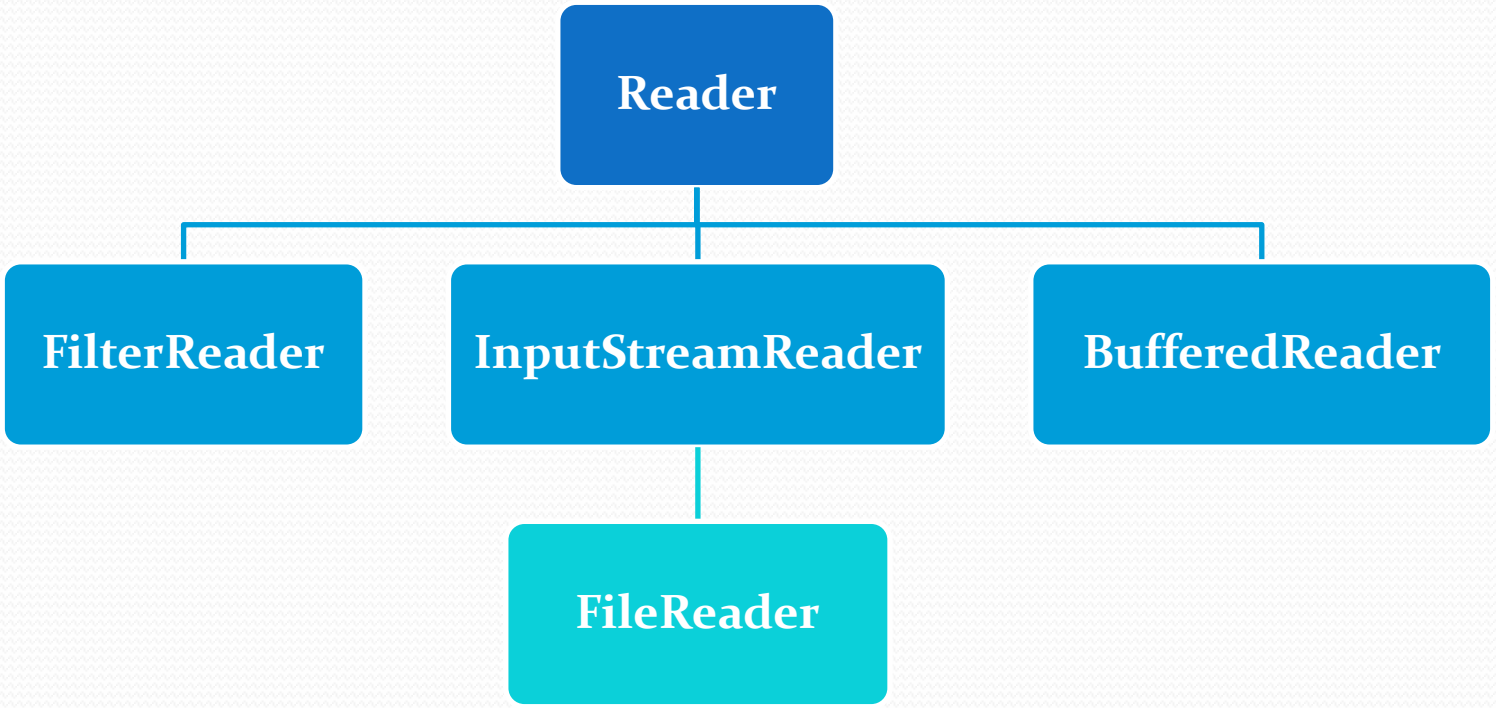
Flux de caractères (textes)

Flux de caractères (textes)

- Ils transportent des données sous forme de caractères : java les gèrent 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.

La classe Reader

- C'est une classe abstraite qui est la classe mère de toutes les classes qui gèrent des flux de caractères en lecture.



Quelques méthodes de Reader (1)

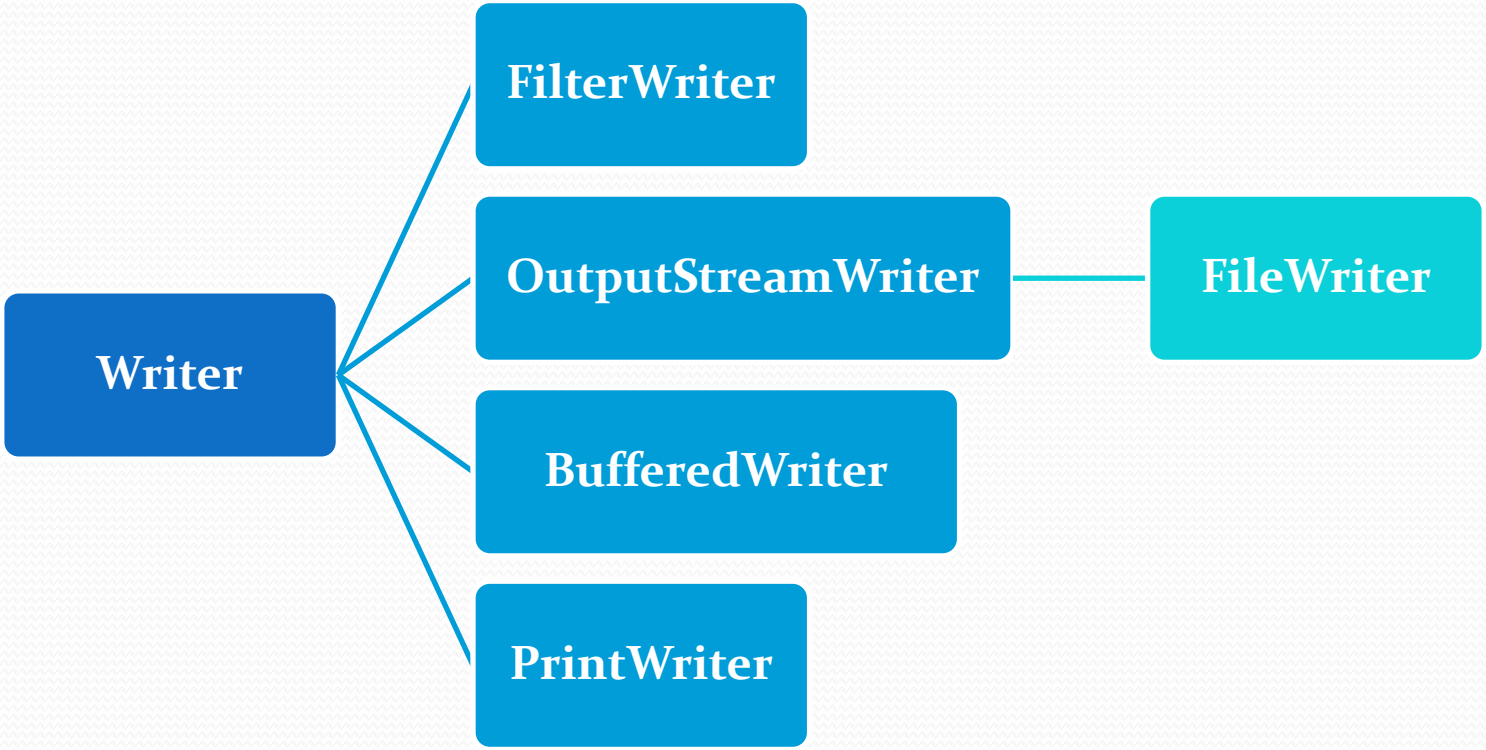
- **boolean ready()**
  - indique si le flux est prêt à être lu
- **close()**
  - ferme le flux et libère les ressources qui lui étaient associées
- **int read()**
  - renvoie le caractère lu ou -1 si la fin du flux est atteinte.
- **int read(char[])**
  - lit plusieurs caractères et les met dans un tableau de caractères

Quelques méthodes de Reader (2)

- **int read(char[], int, int)**
  - Lit plusieurs caractères. Elle attend en paramètre : un tableau de caractères qui contiendra les caractères lus, l'indice du premier élément du tableau qui recevra le premier caractère et le nombre de caractères à lire. Elle renvoie le nombre de caractères lus ou -1 si aucun caractère n'a été lu. Le tableau de caractères contient les caractères lus.
- **long skip(long)**
  - saute autant de caractères dans le flux que la valeur fournie en paramètre. Elle renvoie le nombre de caractères sautés.
- **reset()**
  - retourne dans le flux à la dernière position marquée

La classe Writer

- C'est une classe abstraite qui est la classe mère de toutes les classes qui gèrent des flux de caractères en écriture.



Quelques méthodes de la classe writer

- **close()**
  - ferme le flux et libère les ressources qui lui étaient associées
- **write(int)**
  - écrire le caractère en paramètre dans le flux.
- **write(char[])**
  - écrire le tableau de caractères en paramètre dans le flux.
- **write(char[], int, int)**
  - écrire plusieurs caractères. Elle attend en paramètres : un tableau decaractères, l'indice du premier caractère dans le tableau à écrire et lenombre de caractères à écrire.
- **write(String)**
  - écrire la chaîne de caractères en paramètre dans le flux
- **write(String, int, int)**
  - 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.

Les flux de caractères avec un fichier

- Les classes FileReader et FileWriter permettent de gérer des flux de caractères avec des fichiers.

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

Constructeur	Rôle
FileInputReader (string)	Créer un flux en lecture vers le fichier dont le nom est donné en paramètre
FileInputReader (File)	Céer un flux en lecture vers le fichier donné avec l'objet File

Les flux de caractères en écriture sur un fichier

- Il faut instancier un objet de la classe FileWriter qui hérite de la classe OuputStreamWriter. Cette classe possède plusieurs constructeurs :

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

Constructeur	Rôle
FileWriter(String)	Si le nom du fichier donné en paramètre n'existe, alors le fichier est créé. S'il existe, les données seront écrasées
FileWriter(File)	Même chose mais le fichier est donné avec un objet file
FileWriter(String,Boolean)	Le booléen permet de préciser si les données seront ajoutées au fichier (true) ou écraseront les données existantes (false)

```java
import java.io.*;

public class FluxCaracteres {
    public static void main(String[] args) throws IOException {
        FileReader inputStream = null;
        FileWriter outputStream = null;

        try {
            inputStream = new FileReader("source.txt");
            outputStream = new FileWriter("characteroutput.txt");

            int c;
            while ((c = inputStream.read()) != -1) {
                outputStream.write(c);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

Différence entre les deux programmes

- Les deux programmes FluxBinaire et fluxCaracteres sont similaires, la principale différence est que le programme FluxCaracteres utrilise FileReader et fileWriter our les entrées sorties au lieu de FileInputStream et FileOutputStream.
- De plus, dans les deux programmes, on utilise une variable de type int pour la lecture et pour l'écriture. Mais, dans Fluxcaracteres, l'entier contient la valeur d'un caractères dans ses 16 derniers bits alors que dans Fluxbinaire, l'entier contient la valeur d'un octet dans ses 8 derniers bits.

Les flux de caractères tamponnés avec un fichier

- 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. Le nombre d'opérations est ainsi réduit.
- Les classes BufferedReader et BufferedWriter permettent de gérer des flux de caractères tamponnés avec des fichiers.

Les flux de caractères tamponnés en lecture avec un fichier

- Il faut instancier un objet de la classe BufferedReader. Cette classe possède plusieurs constructeurs qui peuvent tous
- lever une exception de type FileNotFoundException:

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

Constructeur	Rôle
BufferedReader(Reader)	Le paramètre correspond au flux à lire
BufferedReader (Reader, int)	L'entier indique la taille du buffer. Il doit être positif sinon une exception est levée

Les flux de caractères tamponnés en écriture avec un fichier

- Il faut instancier un objet de la classe BufferedWriter. Cette classe possède plusieurs constructeurs

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

Constructeur	Rôle
BufferedWriter()	Le paramètre correspond au flux dans lequel les données seront écrites
BufferedWriter(Writer, int)	L'entier indique la taille du buffer. Il doit être positif sinon une exception est levée

```java
import java.io.*;

public class BufferedFluxCaractere {
    public static void main(String[] args) throws IOException {
        BufferedReader in = null;
        BufferedWriter out = null;
        try {
            in = new BufferedReader(new FileReader("source.txt"));
            out = new BufferedWriter(new FileWriter("destination.txt"));
            int c;

            while ((c = in.read()) != -1) {
                out.write(c);
            }

        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

Accès aux lignes d'un flux texte: La méthode ReadLine()

- Il existe plusieurs méthodes de la classe BufferedReader héritées de la classe Reader qui permettent de lire un ou plusieurs caractères dans le flux. Toutes ces méthodes peuvent lever une exception de type IOException.
- String readline(): permet de lire une ligne 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.

```java
import java.io.*;
import java.util.*;
public class TestReadline {
    public static void main(String[] args) throws IOException {
        BufferedReader in = null;
        String ligne;

        try {
            in = new BufferedReader(new FileReader("source.txt"));

            while ((ligne = in.readLine())!= null)

            {
            System.out.println(ligne);
            StringTokenizer tok = new StringTokenizer(ligne," ");
            int nb = tok.countTokens();
            for( int i=0; i<nb;i++)
            {
             System.out.println(tok.nextToken());
            }
            }
        } finally {
            if (in != null)
                in.close();
        }
    }
}
```

Fichiers

La classe File

- Java dispose d'une classe *File* qui offre des fonctionnalités de gestion de fichiers . Il s'agit d'opérations concernant le fichier dans sa globalité, et non plus chacune des informations qu'il contient.
- C'est ainsi que l'on pourra créer, supprimer ou renommer un fichier ou un répertoire, tester l'existence d'un fichier ou d'un répertoire, obtenir des informations relatives aux protections ou aux dates de modification, lister les noms de fichiers d'un répertoire…

La classe file

Cette classe représente la notion de fichier, indépendamment du système d'exploitation

Un fichier est repéré par un chemin abstrait composé d'un préfixe optionnel (nom d'un disque par exemple) et de noms (noms des répertoires parents et du fichier lui-même)

```java
import java.io.File;

public class TestFile {
public static void main(String[] args) {
//Création de l'objet File
File f = new File("source.txt");
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());
System.out.println("Affichage des lecteurs à la racine du PC : ");
for(File file : f.listRoots())
{System.out.println(file.getAbsolutePath());
try {
//On parcourt la liste des fichiers et répertoires
for(File nom : file.listFiles()){
System.out.print("\t" + nom.getName());
}
System.out.println("\n");
} catch (NullPointerException e) {
}
}
}
}
```

Fichiers à accès direct (1)

- Les fichiers à accès direct permettent un accès rapide à un enregistrement contenu dans un fichier
- La classe RamdonAccessFile encapsule les opérations de lecture/écriture d'un tel fichier. Elle implémente les interfaces DataInput et DataOutput.
- Elle possède deux constructeurs qui attendent en paramètre le fichier à utiliser (sous la forme d'un nom de fichier ou d'un objet de type File qui encapsule le fichier) et le mode d'accès.
- Le mode est une chaîne de caractères qui doit être égal à «r» ou «rw» selon que le mode soit lecture seule ou lecture/écriture.

Fichiers à accès direct (2)

- La classe RandomAccessFile possède de nombreuses méthodes writeXXX() (readXXX()) pour écrire (lire)des types primitifs dans le (du) fichier.
- Pour naviguer dans le fichier, la classe utilise un pointeur qui indique la position dans le fichier ou 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() a comme paramètre un entier long qui réprésente la position, dans le fichier, précisée en octets.
  - La première position commence à zéro.

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

// 5 représente le sixième enregistement 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(" le 5ème entier est :"+monFichier.readInt());
monFichier.close();
}
catch (Exception e) {
e.printStackTrace();
}
}
}
```

La sérialisation

Sérialisation

- C’est l’action de sauvegarder l’état d’un objet dans un fichier. On dit aussi « rendre l’objet persistant »
- Sérialiser un objet c’est transformer l’état (les valeurs des variables d’instance) de l’objet en une suite d’octets et les sauvegarder dans un fichier.
- On peut ainsi conserver l’état de l’objet pour le retrouver ensuite et reconstruire un autre objet avec le même état
- La sérialisation permet de conserver l'état des objets entre deux exécutions d'un programme, ou d'échanger des objets entre programmes.

Qu’est-ce qui est sérialisé ?

- Les valeurs des variables d’instance (pas des variables de classe) des instances sérialisées
- Des informations sur les classes des objets sérialisés ; en particulier :
  - nom de la classe
  - noms, types, modificateurs des variables à sauvegarder
  - des informations qui permettent de savoir si une classe a été modifiée entre la sérialisation et la désérialisation
- Mais le code des classes n’est pas sérialisé !

L'interface Serializable

- Cette interface ne définit aucune méthode mais permet simplement d'indiquer qu'une classe peut être sérialisée.
- Tout classe dont les objets doivent être sérialisée doit implémenter cette interface ou hériter d'une classe implémentant cette interface.
- Si l'on tente de sérialiser un objet qui n'implémente pas l'interface Serializable, une exception NotSerializableException est levée.

La classe ObjectOuputStream

- pour sérialiser des objets on doit instancier la classe ObjectOuputStream en fournissant comme paramètre un objet de type FileOutputStream dans lequel les objets seront stockés.
- Pour écrire dans le fichier en utilise la méthode writeObject() en lui passant en paramètre l'objet à sérialiser
- D’autres méthodes existent pour sérialiser des variables de types primitifs: writeInt(), writeDouble(), writeFloat(), …

La classe ObjectInputStream

- pour récupérer des objets sérialisés on doit instancier la classe ObjectOutputStream en fournissant comme paramètre un objet de type FileOutputStream créé en utilisant le fichier dans lequel les objets ont été stockés
- Un appel à la méthode readObject() retourne l'objet avec un type Object.
- Une conversion de type est nécessaire pour récupérer les objets correctement .
- La classe ObjectInputStream possède de la même façon que la classe ObjectOutputStream des méthodes pour lire des données de type primitifs : readInt(), readDouble(), readFloat(), ...

Exemple de sérialisation

- Nous allons

1. définir une classe Point implémentant l'interface Serializable et créer deux objets
2. Sérialiser les objets créer
3. Les désérialiser afin de les réutiliser

```
import java.io.*;

class Point  {//implements Serializable {

private int x;
private int y;

public Point(int x, int y)
{ this.x = x;
  this.y = y;
}
public void afficher()
{
System.out.println(" les coordonnées du point sont :
  "+ x+" "+y);
}

}
```

```java
import java.io.*;

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

//Ecriture des objets
ObjectInputStream in;
ObjectOutputStream out;
try {
out = new ObjectOutputStream(
new BufferedOutputStream(
new FileOutputStream(
new File("Point.dat"))));

//écriture les objets dans un fichier
out.writeObject(new Point(1,2));
out.writeObject(new Point(3,4));
out.writeObject(new Point(5,6));

//Fermer le flux
out.close();

//Récupération des objets stockés

in = new ObjectInputStream(
new BufferedInputStream(
new FileInputStream(
new File("point.dat"))));
try {

System.out.println("Affichage des points:");
System.out.println("************************\n");
((Point)in.readObject()).afficher();
((Point)in.readObject()).afficher();
((Point)in.readObject()).afficher();
} catch (ClassNotFoundException e) {
e.printStackTrace();
}
in.close();
} catch (FileNotFoundException e) {
e.printStackTrace();
} catch (IOException e) {
e.printStackTrace();
}
}
}
```

Le mot clé transient

- Supposons que nous ayons une classe cercle ayant comme attributs un champs centre de type texte et un attribut rayon de type réel.
- La sérialisation des objets de type cercle ne peut se faire que si la classe Point est elle-même sérialisable.
- Dans le cas où on ne veut pas que la classe Point soit sérialisable, on doit utiliser le mot clé **transient** afin de le préciser que cette variable ne rentre pas dans le processus de sérialisation/déserialisation.
- Dans ce cas, lors de la sérialisation, les objets déclarés transient sont ignorés et lors de la désérialisation, les champs transient sont initialisés avec la valeur null.

```java
import java.io.*;

public class Cercle implements Serializable

{
private transient Point centre;
private float rayon;

public Cercle (int x, int y, float r)
{
centre = new Point (x,y);
rayon = r;
}
 public void afficher ()
{//centre.afficher();

System.out.println(" rayon :" + rayon);
}

}
```

Java.nio

Java.nio (1)

- nio signie New I/O. Ce package a été créé an d'améliorer les performances sur le traitement des flux et des fichiers.

- Ce package permet de lire les données d'une façon différente. Il utilise les possibilités avancées du système d'exploitation hôte pour optimiser les entrées-sorties et offrir plus de fonctionnalités
- Alors que les objets du package java.io traite les données par octets. Les objets du package java.nio sont traités par blocs de données ce qui accélère la lecture

Java.nio (2)

- Deux nouvelles classes: les **Channels** et les **Buffers**.
- Les channels sont en fait des flux qui sont amenés à travailler avec un buffer dont on définit la taille.
- Ainsi, lorsque on ouvre un flux vers un fichier avec un objet FileInputStream, on peut récupérer un canal vers ce fichier et le combiner à un buffer pour lire le fichier encore plus vite qu'avec un BufferedInputStream !

- Exemple de copie de fichier avec nio (sous eclipse)

Utilisation

- Ce package est le surtout utilisé pour traiter les flux circulant sur les réseaux. Pour améliorer les performances des serveurs surchargés.
- Il est un peu plus complexe à utiliser que le package io.
- A utiliser en cas de nécessité