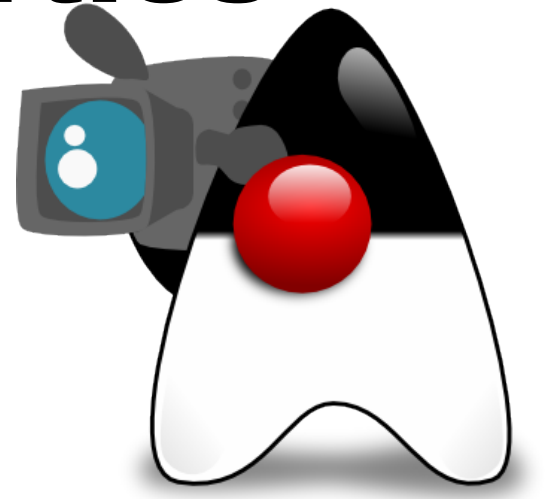


INPUT OUTPUT

Gestion d'entrées/sorties



Plan du cours



- Introduction: Expliquer les flux (streams)
- Introduction: `java.io` → `java.nio`
- La classe `File`
- Flux d'octet : Byte streams
- Flux de caractère: Char streams
- Flux de buffer : Buffered streams
- Position dans un fichier
- Retour sur les exceptions
- Flux d'objet : Object streams (serialization)
- Travailler avec le format Zip (compression, décompression)

Objectif du cours

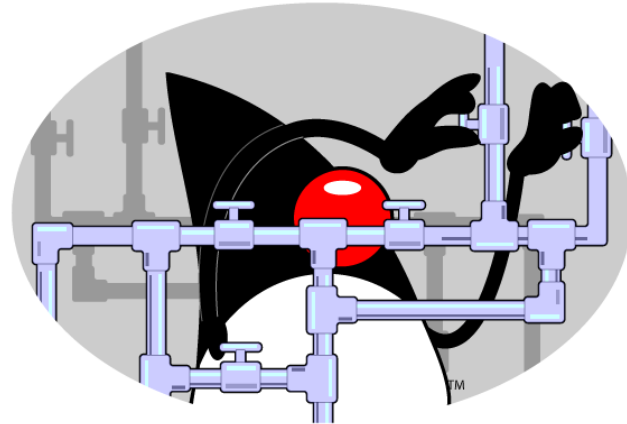


A la fin de cours vous serez capable de:

- Lire & écrire différents types de flux (stream)
- Sérialiser désérialiser des objets

Input Output

INTRODUCTION





Objectif de la gestion de fichiers :

- Pouvoir conserver des données en mémoire de **façon durable**.
- Pour l'instant nos données ne sont disponibles que **pendant** l'exécution du programme.
- Un fichier permet de conserver ces données lorsque notre programme s'arrête.

Identification /Chemin

Identification des fichiers :

Un fichier s'identifie par des attributs de fichier **Nom**, Identificateur interne, Type, Adresse physique, Taille, Permissions d'accès, Dates... **et le chemin pour y accéder.**

Le fichier fait partie d'un arbre de fichiers (arborescence).

Cet arbre est constitué de fichier et de répertoire.

Un répertoire (appelé également dossier ou folder en anglais) est un objet informatique pouvant contenir des fichiers ou des sous dossier et ainsi de suite dans les sous dossier formant cette arborescence.

Le répertoire racine en anglais (root directory) est l'entité de plus bas niveau car elle peut contenir des fichiers ou des répertoires mais ne peut pas se trouver elle-même dans un répertoire.

Elle se traduit généralement par la lettre C: ou D:

Dans cet arbre de fichier pour retrouver répertoire, fichier, l'utilisateur parcourt un chemin:

Chemin absolu, relatif

Le chemin qui permet d'identifier un fichier depuis la racine de l'arbre, se qualifie de chemin absolu tandis que le chemin à partir du répertoire courant (répertoire dans lequel sont rangés les fichiers sur lesquels l'utilisateur agit) s'appelle chemin relatif.



2 principaux types de fichiers :

Fichier texte : fichier organisé sous la forme d'une suite de lignes, chacune étant composée d'un certain nombre de caractères et chaque ligne se terminée par '\n'.

Fichier binaire : suite d'octets, pouvant représenter toutes sortes de données.

A noter : La J.V.M exécute vos programmes Java à partir du fichier binaire '.class'
(fichier contenant le byte code de votre application
Flux d'octet binaire)





buffer

- Pour un programme l'accès à un fichier n'est pas direct il se fait par l'intermédiaire d'une mémoire-tampon (**buffer**) limitant le nombre d'accès aux périphériques (disque...)
- Le buffer est vidé:
 - Quand il est plein
 - Quand il contient '\n'



Généralités

Pour manipuler un fichier, un programme a besoin d'informations :

- l'adresse du fichier dans la mémoire-tampon
- le mode d'accès au fichier (lecture ou écriture)
- ...

Flux d'entrée/sortie Input/Output

- Un flux (stream) est une séquence continue de données.
- I \rightarrow in pour la lecture de flux provenant d'une source :
input source
- O \rightarrow out pour l'écriture de flux vers une destination:
output destination
- Sources et destinations peuvent être:
 - périphériques d'entrée et sortie,
 - d'autres programmes,
 - tableau en mémoire,
 - fichier

Utilisation

- **modes d'ouverture possibles :**

- **Lecture:** lire le contenu du fichier

Condition: Le fichier doit exister auparavant.

Existe une condition d'arrêt de lecture quand il n'y a plus rien à lire
(la fin de fichier E.O.F)

- **Écriture:**

Par défaut: Si fichier existe le contenu est effacé, sinon fichier créé.

En mode Ajout,

si le fichier existe, écriture se fait à la fin de celui-ci sans effacer son contenu (mode par défaut)

mais il est possible de positionner un curseur à une position précise dans le fichier autant pour la lecture que l'écriture

Différents type de données

- Le flux est utilisé pour échanger des données, il supporte de nombreux types de données :
 - des valeurs primitives,
 - des objets fournis par la J.D.K
 - nos propres objets (sérialisation)
- Les classes qui fonctionnent avec des flux de fichiers sont situées dans les paquetages:
 - java.io.
 - java.nio.

Package java.io

- Le Package **java.io** fournit la plupart des classes et méthodes nécessaires à la création, lecture, écriture et traitement des flux I/O.
- 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 d'octet
 - les flux de communication et les flux de traitement
 - les flux avec ou sans buffer

Package java.nio

Blocking IO wait for the data to be write or read before returning.It means when the thread invoke a write() or read(), then the thread is blocked until there is some data available for read, or the data is fully written.

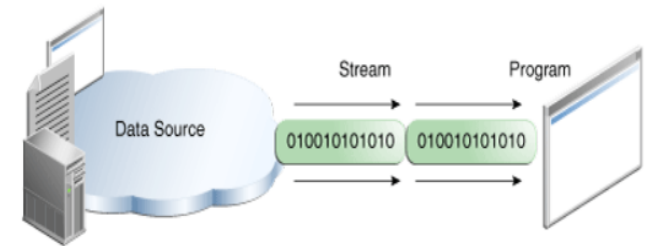
Nio → non blocking I/O operation

Non blocking IO does not wait for the data to be read or write before returning. Java NIO non- blocking mode allows the thread to request writing data to a channel, but not wait for it to be fully written. The thread is allowed to go on and do something else in a mean time.

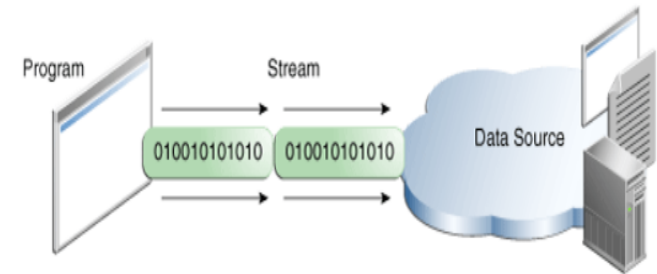
Utilisation

Tous flux utilisent le même principe simple d'utilisation :

- Ouverture d'un flux
- Lecture avec flux d'entrée (input stream)
- Ecriture avec flux de sortie (output stream)
- Fermeture du flux



Reading information into a program.



Writing information from a program.

Package java.nio

L'API NIO 2 a été ajoutée au JDK dans la version 7 de Java SE. NIO 2 est une API plus moderne et plus complète pour l'accès au système de fichiers. que la classe File et la très ancienne API IO.

NIO 2 propose d'étendre les fonctionnalités relatives aux entrées/sorties : l'utilisation du système de fichiers de manière facile et les lectures/écritures asynchrones.

La nouvelle API de gestion et d'accès au système de fichiers est contenue dans le package java.nio.file et ses sous-packages.

NIO2 facilite la mise en oeuvre de fonctionnalités courantes d'entrées/sorties sur un système de fichiers.

Retour sur les exceptions

Rappel : la manipulation de fichier se doit de gérer les exceptions :

erreurs surveillées (IO exception)

Conséquence : obligation d'utiliser

→ les bloc : catch, try, finally

→ la clause throws



méthode close lève exception IOException

```
FileInputStream fis = null;
FileOutputStream fos = null;
try
{
    fis = new FileInputStream("input.txt");
    fos = new FileOutputStream("output.txt");
    // ...
}
catch (IOException e) {
    // ...
}
finally
{
    if(fis != null) {
        //la méthode close() lève une exception IOException
        try { fis.close(); }
        catch (IOException e) { /* ... */ }
    }
}
```

Automatic Resource Management

Des ressources comme des fichiers, des flux, des connexions, ... doivent être fermées explicitement par le développeur (voire diapo précédente)

Depuis Java 7, l'instruction `try with resource` permet de définir une ressource qui sera automatiquement fermée.

Ce mécanisme se nomme Automatic Resource Management (A.R.M) .

```
FileInputStream fis = null;
FileOutputStream fos = null;
try {
    fis = new FileInputStream("input.txt");
    fos = new FileOutputStream("output.txt");
    // ...
}
catch (IOException e) {
    e.printStackTrace();
}
```

n.i.o plus complète (exception)

exemple: Créer un fichier en gérant si le fichier existe déjà

```
Path ch = Paths.get("java.txt");
```

```
try {
```

```
    Files.createFile(ch);
```

```
} catch (FileAlreadyExistsException fae)
```

```
    System.err.format("file named %s already exists%n", ch);
```

Interface Path

Les classes Paths & Files contiennent des méthodes statiques

'**Paths**' pour manipuler un chemin

'**Files**' pour des opérations sur des fichiers

<https://docs.oracle.com/javase/tutorial/essential/io/file.html#channels>

java.io → java.nio

Avec java.io plusieurs méthodes didn't throw exceptions donc pas de message d'erreur explicite.

Because the Java implementation of file I/O has been completely re-architected in the Java SE 7 release, you cannot swap one method for another method mais

nio fournit la méthode toPath, pour une conversion java.io → java.nio

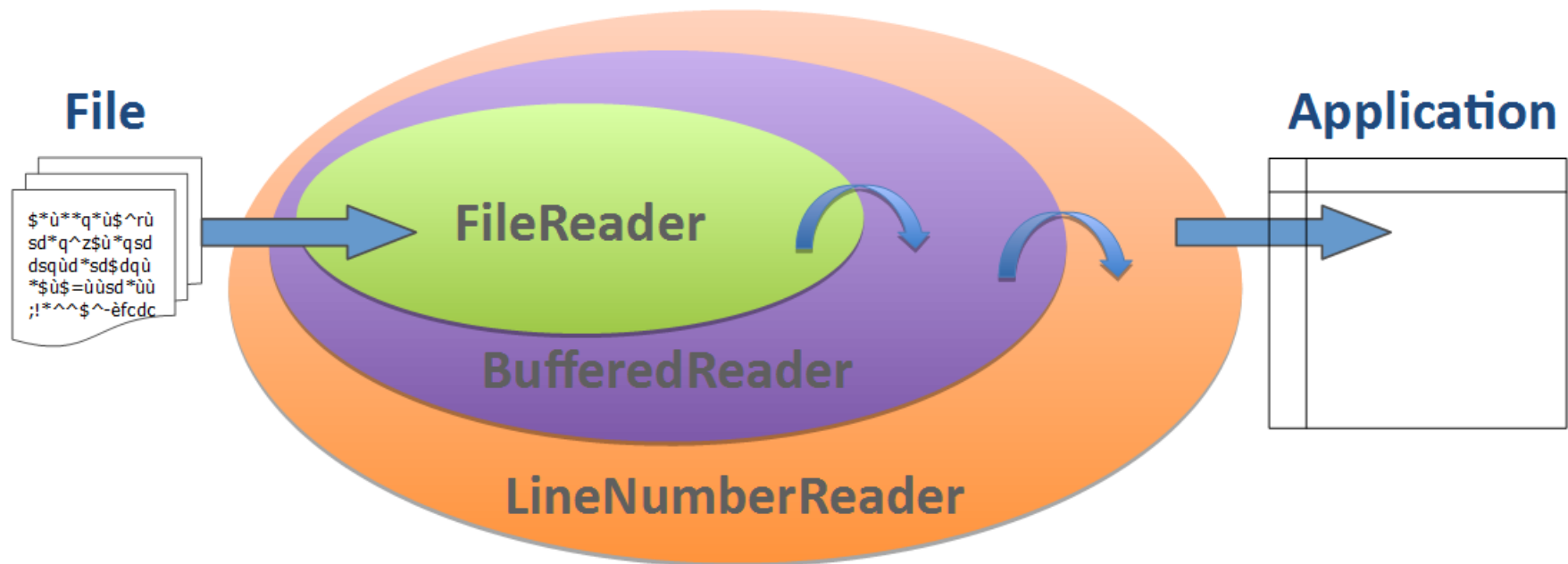
```
File outputFile = new File("javax.txt");
try {
    outputFile.createNewFile();
} catch (IOException ioe) {
    System.out.println(ioe.getMessage()); // pas d'exception si file déjà créer
}
```

```
Path outputPath = outputFile.toPath();
try {
    Files.createFile(outputPath);
}
catch (FileAlreadyExistsException fae) {
    System.err.format("file named %s already exists%n", outputPath);
}
```

file named javax.txt already exists

Un aperçu des classes

- le IO (Input/Output) system:



Input Output

La classe `java.io.File`

File operation

Indépendant de l'O.S

Comme la notion de fichier et leur organisation dépendent de l'O.S, en Java (langage multiplateforme) un objet `File` est la représentation **abstraite** d'un fichier, d'un répertoire...

- Les constructeurs disponibles:
 - `File(String pathname)`
 - `File(String parent, String fileName)`
 - `File(File parent, String fileName)`
 - `File(URI uri)` uniform ressource identifier = u.r.l

Le constructeur de la classe `File` crée en mémoire une instance de type `file` qui pointe sur le fichier réel.

Construction d'un fichier : Exemple

Construction et chemin pour le fichier :

```
//Le fichier existera dans le répertoire courant
File Monfichier = new File( "monFichier.txt");
// Le fichier existera dans le répertoire directory
File Directory = new File("myDirectory");
if (Directory.mkdir())
    System.out.println("le repertoire directory existe");
File fichier1 = new File(Directory, "monFichier.txt");

// Monfichier existe quand un flux (sortant) écrit dedans.
System.out.println(" fichier exist ? : "+Monfichier.exists() );
FileWriter out = new FileWriter(Monfichier);
out.write("bonjour evrybody");
System.out.println("fichier exist ? : "+Monfichier.exists() );
}
```

 Console 

```
<terminated> MainCoursClassFileCreateFile [Jav
fichier: exist ? : false
fichier: exist ? : true
```

Les principales méthodes:

- boolean **createNewFile()** : créer un new file et teste si création réussie
- boolean **canRead()** : test si lecture possible
- boolean **canWrite()** : test si ecriture possible
- boolean **isFile()** : teste si c'est un fichier
- boolean **isDirectory()** : teste si c'est un repertoire
- boolean **mkdir()**
- boolean **makedirs()**
- boolean **exists()**
- long **length()** : retourne la longueur en byte du file

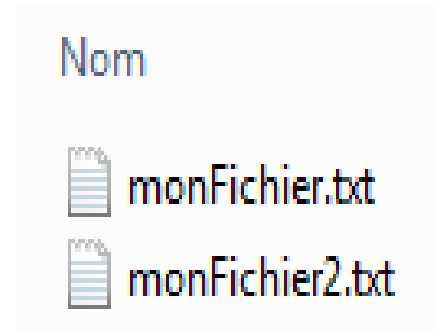
Exemple complet

```
File Directory = new File("myDirectory");
if (Directory.mkdir())
    System.out.println("le repertoire directory existe");
//ajoute le fichier «monFichier.txt » dans le repertoire
File fichier = new File(Directory, "monFichier.txt");
FileWriter out = new FileWriter(fichier);
out.write("bonjour evrybody");//ecriture dans le file
out.close();
System.out.println("longueur: "+fichier.length());
System.out.println("chemin complet: "+fichier.getAbsolutePath());;
//ajoute un autre fichier dans le repertoire
File fichier2 = new File(Directory,"monFichier2.txt");
System.out.println("fichier2 exist ?: "+fichier2.exists() );

File [] tabFile ;

tabFile = Directory.listFiles();
for (int i = 0; i < tabFile.length;i++)
    System.out.println(tabFile[i]);
```

Ordinateur ► Systeme (C:) ► myDirectory



Input Output

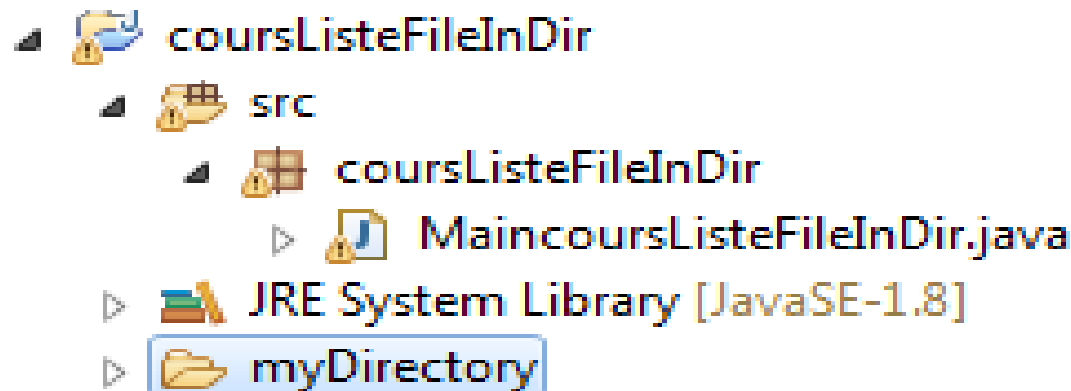
Interface Path /class Paths

`java.nio.file`

présentation

- introduced in the Java SE 7 release, is one of the primary entrypoints of the `java.nio.file` package
- is a representation of a path in the file system.
- A `Path` object contains the file name and directory list used to construct the path,
- used to examine, locate, and manipulate files.

```
Path dir = Paths.get("myDirectory");
```



Exemple: Lister les fichiers d'un répertoire

// IO → object file

```
String [] tabFile ;  
File myDir = new File("myDirectory");  
tabFile = myDir.list();  
for (String nameFile:tabFile )  
    System.out.println(nameFile);
```

// NIO → object path avec DirectoryIteratorException

```
Path dir = Paths.get("myDirectory");  
try (DirectoryStream<Path> stream = Files.newDirectoryStream(dir))  
{  
    for (Path file: stream)  
        System.out.println(file.getFileName());  
}  
catch (IOException | DirectoryIteratorException x) {  
    System.err.println(x);  
}  
} // toujours une meilleure gestion des erreurs avec NIO
```

Flux d'octet (byte streams)

présentation

Un programme java utilise les flux d'octets pour réaliser input & output d'octet (8 bits = 1 byte).

input → lecture provenant du fichier

output → écriture en destination du fichier

Les classes pour utiliser des flux byte stream dérivent de:

- **InputStream**
- **OutputStream.**

Un octet

00100010	00101001	00100010
00101101	00100010	00010101
11010101	00101010	00101001
...		

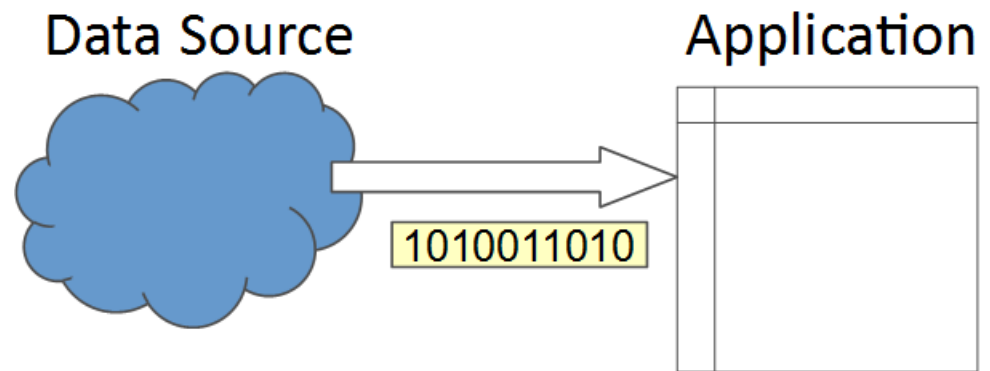
Fichier

Flux d'octet (byte streams)

Lecture

flux d'octets en entrée avec InputStream

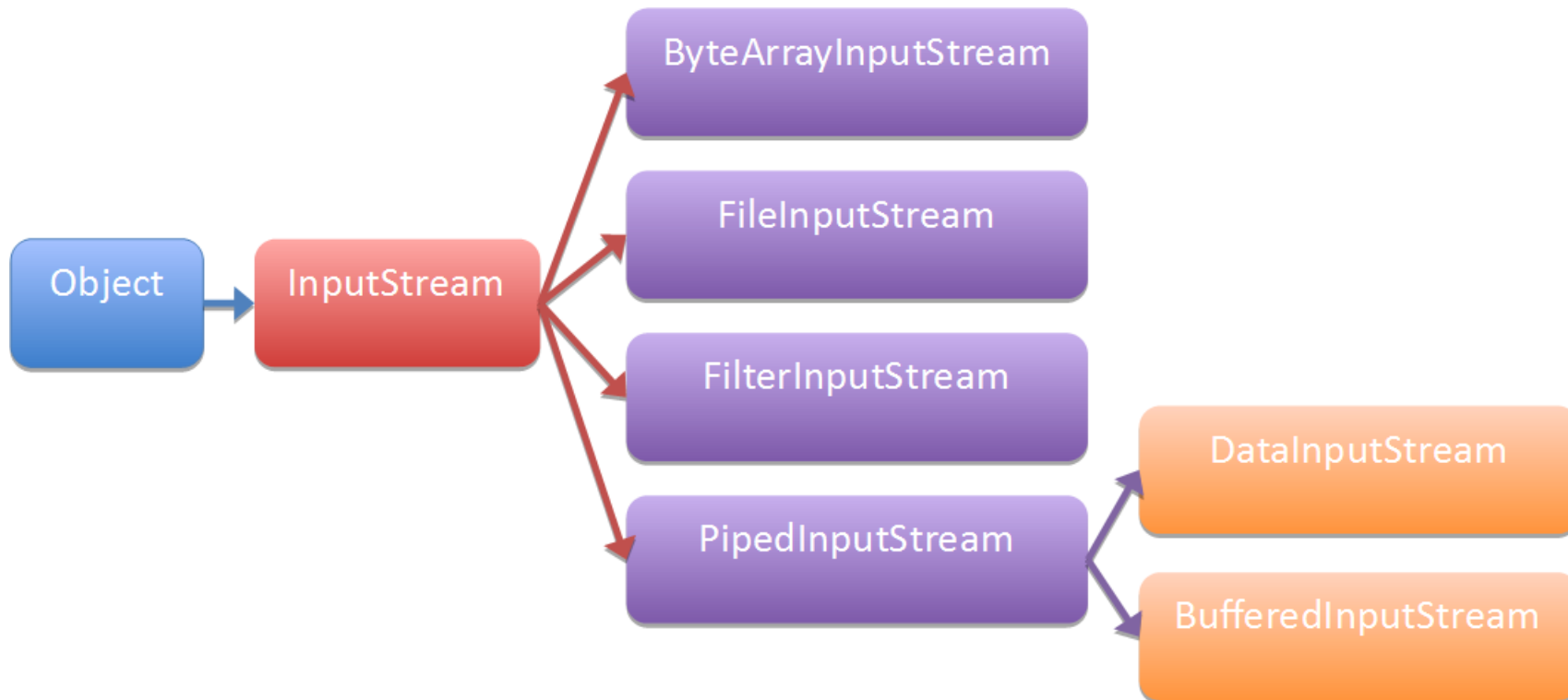
- **Flux d'entrée:** du fichier vers l'application



- La classe **InputStream**:
 - Classe **abstraite** pour lire les flux d'octets en entrée

Classes héritant d'`InputStream`

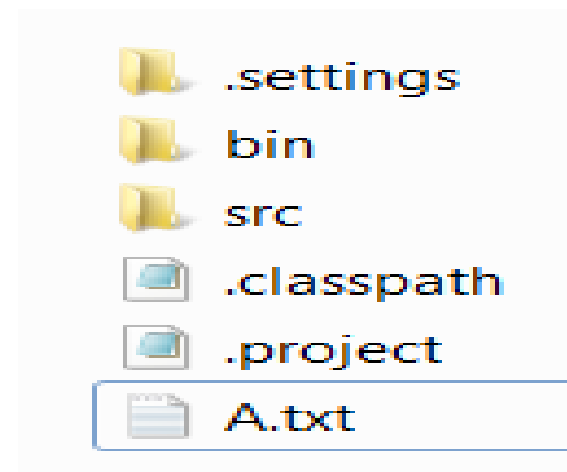
- pour la lecture



La classe fille FileInputStream

Class name	Constructor's argument	Specificity
FileInputStream	File file FileDescriptor fileDesc String fileName	récupère les octets lus à partir d'un fichier.

`FileInputStream fis = new FileInputStream("A.txt");`
Le fichier A.txt doit exister dans le répertoire courant
(chemin relatif)



méthode **read**

La super Classe abstraite **InputStream** contient la méthode abstraite **read**

La méthode, `int read()`, est redéfinie et utilisable dans les sous classes dont la classe **FileInputStream**

- Chaque appel renvoie un octet extrait du flux sous forme d'un entier `[0...255]` correspondant à un caractère de la table a.s.c.i.i
- Retourne `-1` s'il n'y a plus d'octets à lire
→ fin de fichier / End Of File (E.O.F)

méthode **read** : exemple

```
try {  
    FileInputStream fis = new FileInputStream("A.txt");  
    System.out.println(fis.read()); → 65  
    System.out.println(fis.read()); → -1  
} catch (IOException e) { e.printStackTrace(); }
```

Console

65
-1

lecture intégrale du fichier avec une boucle while :

```
int c;
```

```
while ((c = fis.read()) != -1)
```

```
    System.out.println("lecture intégrale octet par octet");
```



Surcharge de la méthode read

- **int read(byte[] *tabByte*):**
 tabByte est le tableau (de type byte) stockant les octets lus.
 Return the number of byte read sous forme d'entier
- **int read(byte[] *b*, int *off*, int *len*):**
 From the offset *off* and length *len*



Byte streams : inputStream class

exemple read(byte[] tabByte)

```
File file = new File("AAAAA.txt");  
byte[] buffer = new byte[(int)file.length()];  
FileInputStream fis;  
fis = new FileInputStream("AAAAA.txt");  
fis.read(buffer);
```

```
for(int i=0; i<buffer.length; i++)  
    System.out.println(buffer[i]);  
//info binaire transformée en texte  
String s = new String(buffer);  
System.out.println(s);
```

 Console 

 Console 
<terminated> m

65
65
65
65
65

info binaire transforme en texte:

AAAAA

Byte streams : inputStream class

La méthode close()

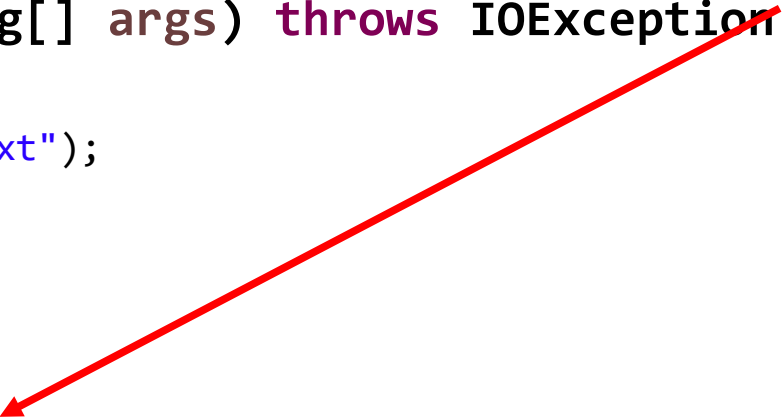
void close(): Close the stream

Toujours fermer le flux en codant un bloc finally
→ garantie que

- le flux sera fermé même si une erreur se produit,
- les ressources utilisés sont libérées

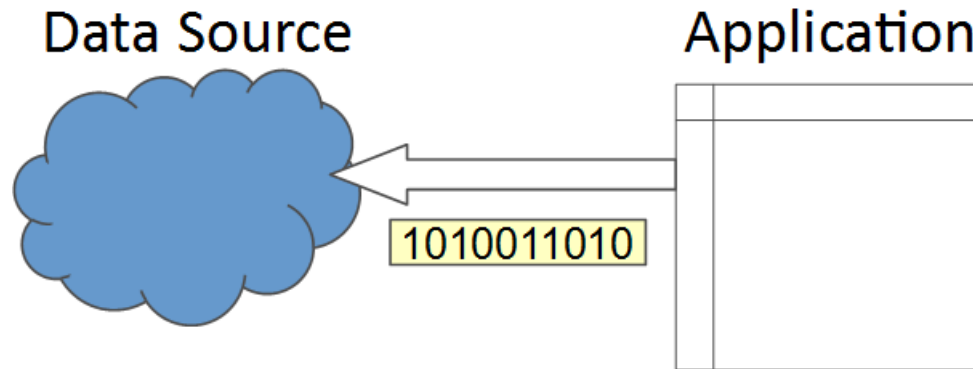
évitant ainsi les fuites mémoires (memory leak)

```
public static void main(String[] args) throws IOException {  
    try {  
        fis = new FileInputStream("A.txt");  
        ...  
        while ((c = fis.read()) != -1)  
            ...  
    }  
    finally  
    {  
        if (fis != null) fis.close();  
    }  
}
```



Ecriture flux d'octet

Flux de sortie avec la classe outputStream



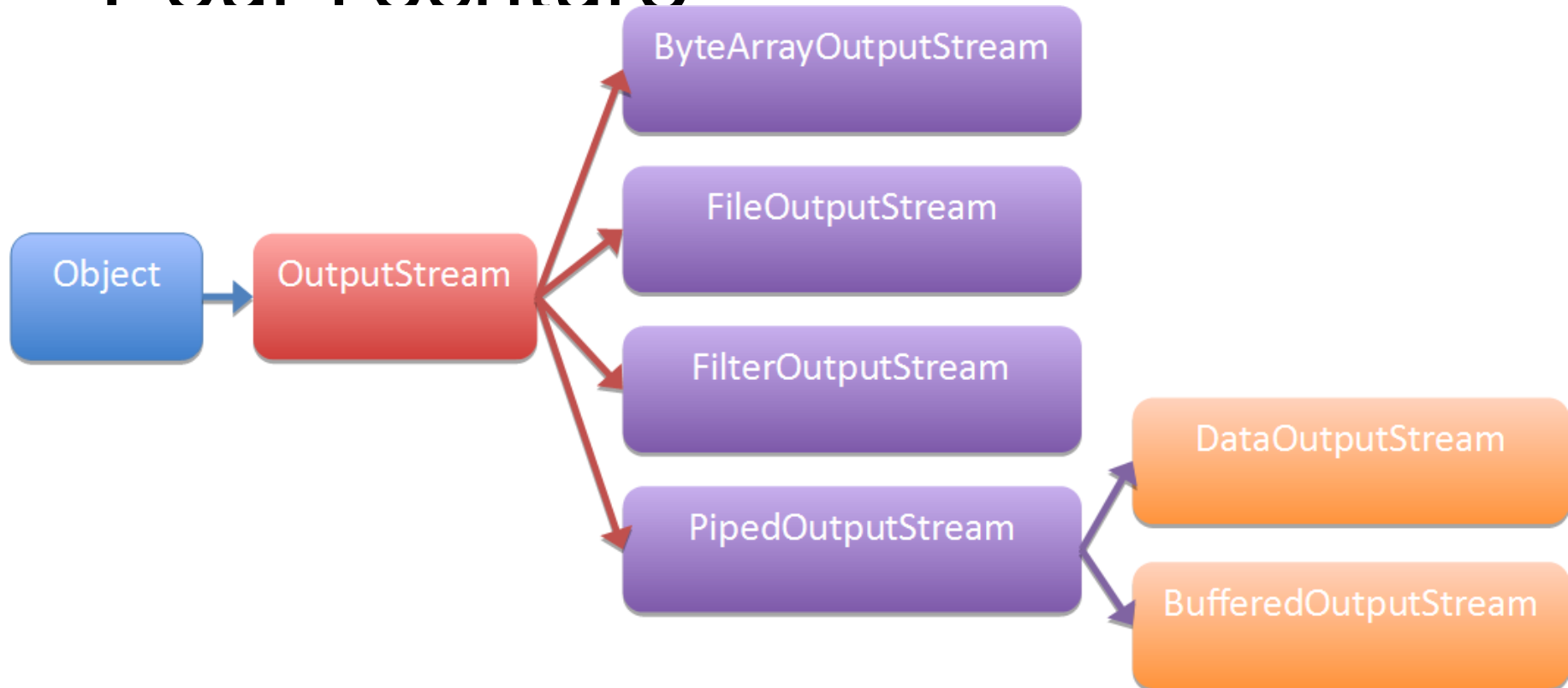
La classe **OutputStream**

OutputStream:

- Classe **abstraite** pour écrire sur les octets sur le flux
- Since JDK 1.0
- Super-classe **OutputStream** de toutes les classes qui representent des flux (d'octets) en sortie*

Classe héritant d'OutputStream

- Pour l'écriture



classe **FileOutputStream**

- Aperçu :

Class name	Constructor's argument	Specificity
FileOutputStream	File file FileDescriptor fileDesc String fileName	Ecrit dans le fichier

- Si le fichier n'existe pas il est créer

```
FileOutputStream out = new FileOutputStream("AOneAgain.txt");
```

Méthodes disponibles:

- **void write (int onebyte)**
 - écrit un octet à la fois
- **void write(byte[] b)**
 - écrit **b.length** octets
- **void writre(byte[]b, int off, int len)**
 - écrit **une partie du tableau b** d'octets
à partir de l'offset ***off*** et de longueur ***len***
- **void close()**
 - ferme le flux

void write(byte[] b):

```
FileOutputStream out = null;  
String WriteToFile = "String écrit dans le file";  
byte[] buffer = WriteToFile.getBytes();  
out = new FileOutputStream("WriteFluxOctet.txt");  
out.write(buffer);  
WriteToFile =  
"ajout dans le file a la suite du contenu existant car flux pas  
ferme";  
buffer = WriteToFile.getBytes();  
out.write(buffer);  
out.close();
```

 WriteFluxOctet.txt - Bloc-notes

```
String écrit dans le file ajout dans le file a la suite du contenu existant car flux pas ferme
```

Pour écrire à la fin de fichier sans écraser le contenu existant ouvrir le fichier en mode ajout:

```
out = new FileOutputStream("WriteFluxOctet.txt",true );
```



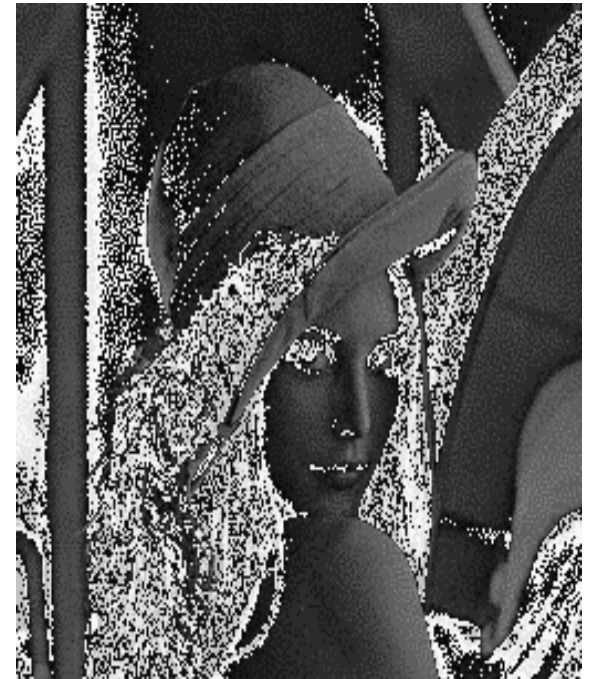

void write (int onebyte)

```
public static void main(String[] args) throws IOException {
    FileInputStream in = null;
    FileOutputStream out = null;
    try {
        in = new FileInputStream("A.txt");
        out = new FileOutputStream("AOneAgain.txt");
        int c; // la méthode read retourne un entier
        while ((c = in.read()) != -1) { out.write(c); }
    }
    catch (FileNotFoundException e) {
        System.out.println("File not found");
    }
    catch (IOException e) {
        System.out.println("Unable to write");
    }
    finally {
        if (in != null) { in.close(); }
        if (out != null) { out.close(); }
    }
}
```



Exercice voir support pdf exercice file

- Comme nous savons maintenant gérer des flux d'octets :
 - écrivons un programme pour modifier les pixels d'une image!

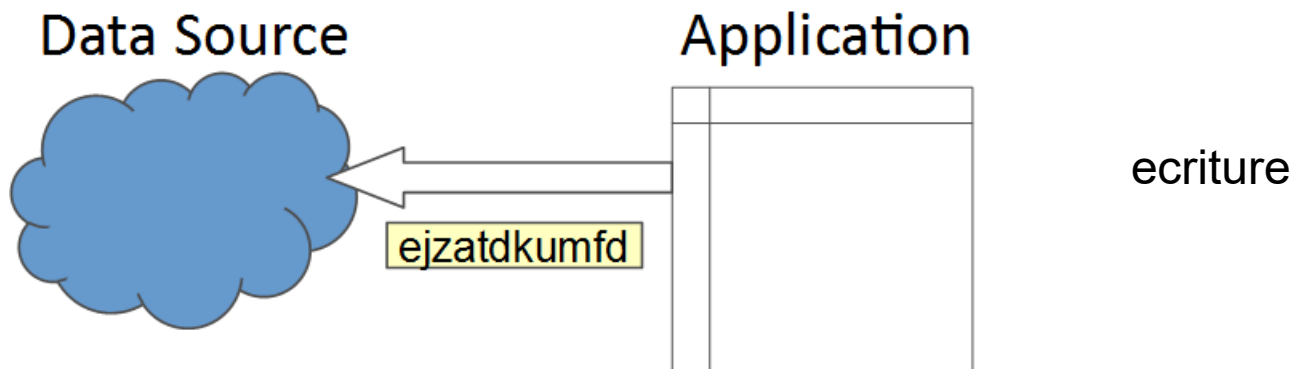
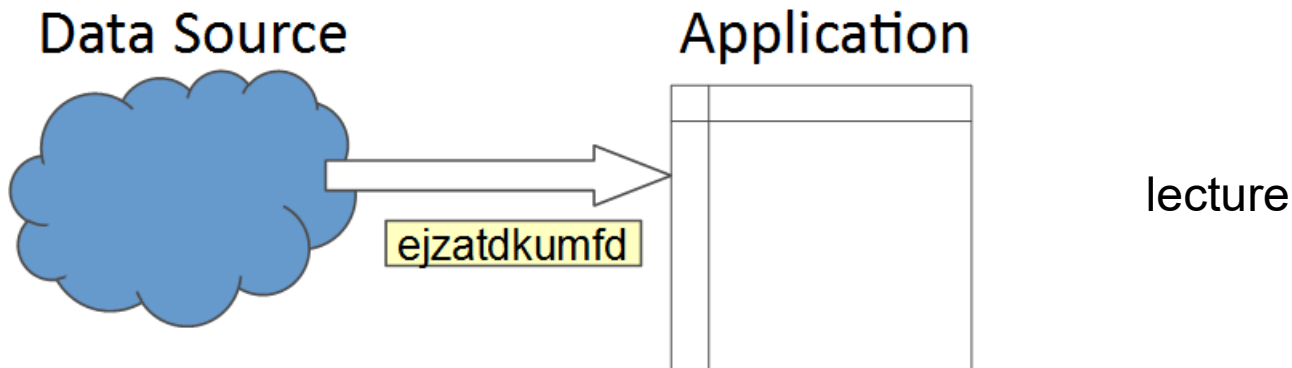


FLUX DE CARACTERE

(character streams)

Présentation

Avec ces flux l'information élémentaire est le caractère (Java les gèrent avec le format Unicode → codage du type char sur 2 octets 16 bits).



Super classes Reader & Writer

- Reader & Writer are the evolution of Input & Output Stream

The character stream uses the byte stream to perform the physical I/O (work in progress...)

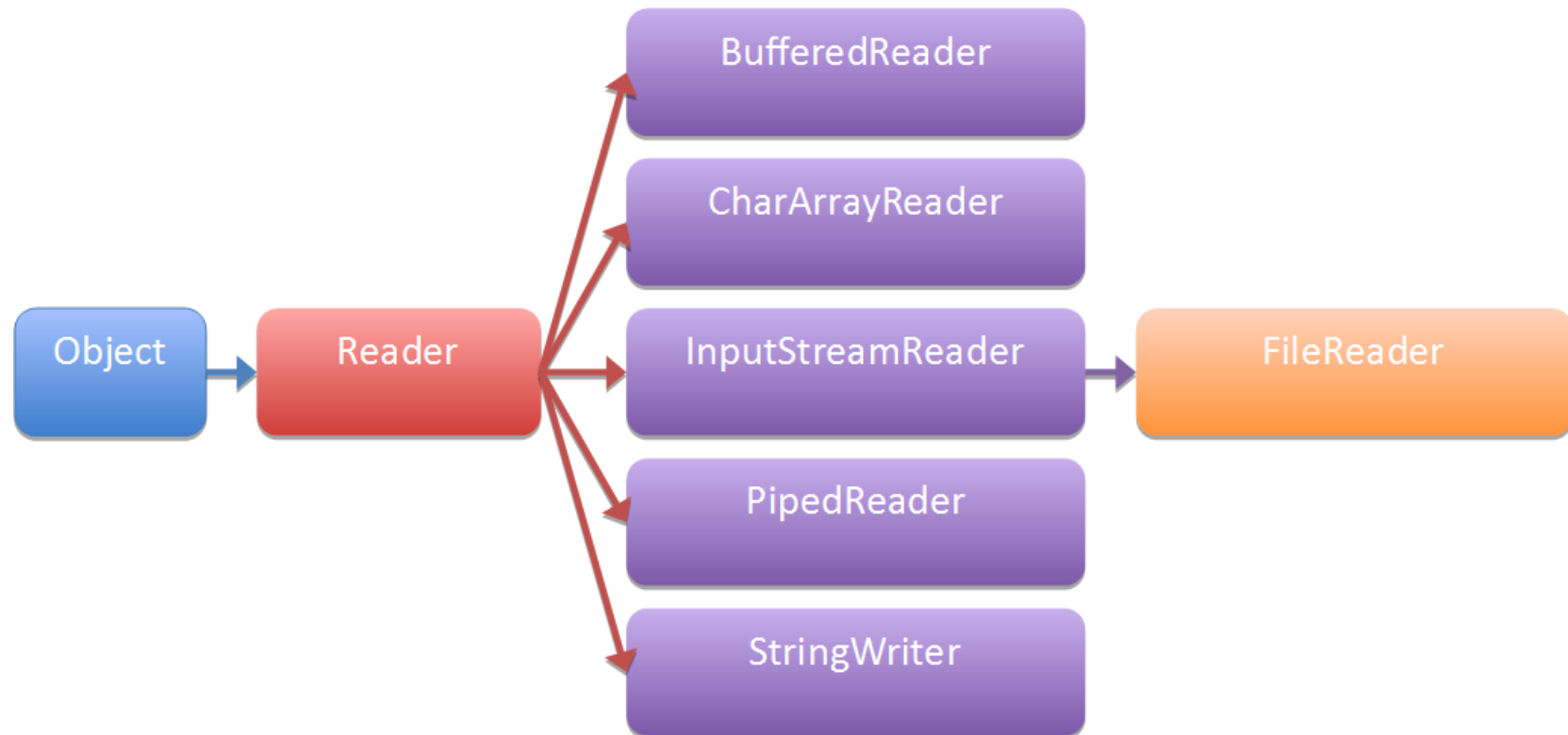
- Benefits:
 - Internationalization (Unicode support)
 - Rapidity

...

FLUX DE CARACTERE

LA CLASSE `java.io`.**READER**

Arborescence



Flux de caractere : la classe reader

Super classe reader

- Classe abstraite, super-classe de toutes les classes lisant des flux de caractères.
- Contient méthode abstraite :
 - read(char[] cbuf, int off, int len)
 - close()
- Fournit les méthodes :
 - int read() → **lit le prochain caractere sur le flux et le retourne sous forme d'int**
 - int read(char[]) → stocke les caracteres lus dans un tableau

These methods acts quite like InputStream ones

conversion automatic des données du type binaires vers le type character

<https://docs.oracle.com/javase/8/docs/api/java/io/Reader.html>

Classe concrète FileReader

lecture fichier texte:

- caractère par caractère
→ méthode read()
- ligne par par ligne
→ méthode readLine().
→ avec utilisation d'un
BufferedReader

Les buffers accélèrent le traitement des fichiers.

La classe FileReader

Lecture d'un fichier texte caractère par caractère
– à l'aide de la méthode read(),

```
Reader in = new FileReader("dico.txt");
int data = 0;

while((data = in.read()) != -1)
    System.out.print((char)data);

in.close();
```

```
//InputStreamReader bridge entre byte et character stream
Reader in = new InputStreamReader(new FileInputStream("dictionnaire.txt"));
```

La classe FileReader

- Lecture d'un fichier texte ligne par ligne
 - avec la méthode `readLine()`
 - présence d'un buffer obligatoire

```
BufferedReader inBuffer;  
inBuffer = new BufferedReader(new FileReader("dico.txt"));  
int nbligne = 0;  
while (inBuffer.readLine() != null)  
    nbligne++;  
System.out.println("nb line in file: "+nbligne);  
inBuffer.close() ;
```



Sous classes utilisant les méthodes

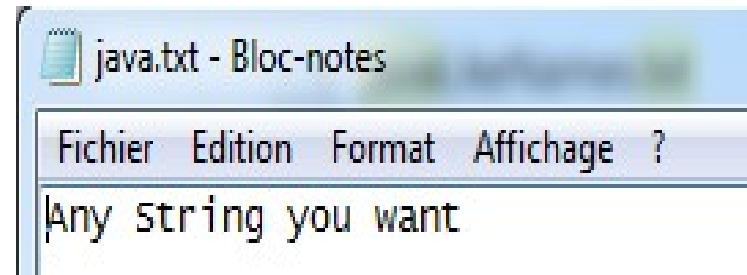
Class name	Constructor's argument	Specificity
InputStreamReader	InputStream is InputStream is, Charset cs	Read bytes and decodes them into characters
FileInputStream	File file FileDescriptor fileDesc String fileName	Efficient reading of chars, arrays and lines

```
InputStreamReader reader =  
    new InputStreamReader(new FileInputStream("dico.txt"));  
int c = 0;  
while((c = reader.read()) != -1)  
{  
    System.out.print((char)c);  
}  
reader.close();
```

Utilisation des sous classes

InputStreamReader lit des bytes et les transforme en char

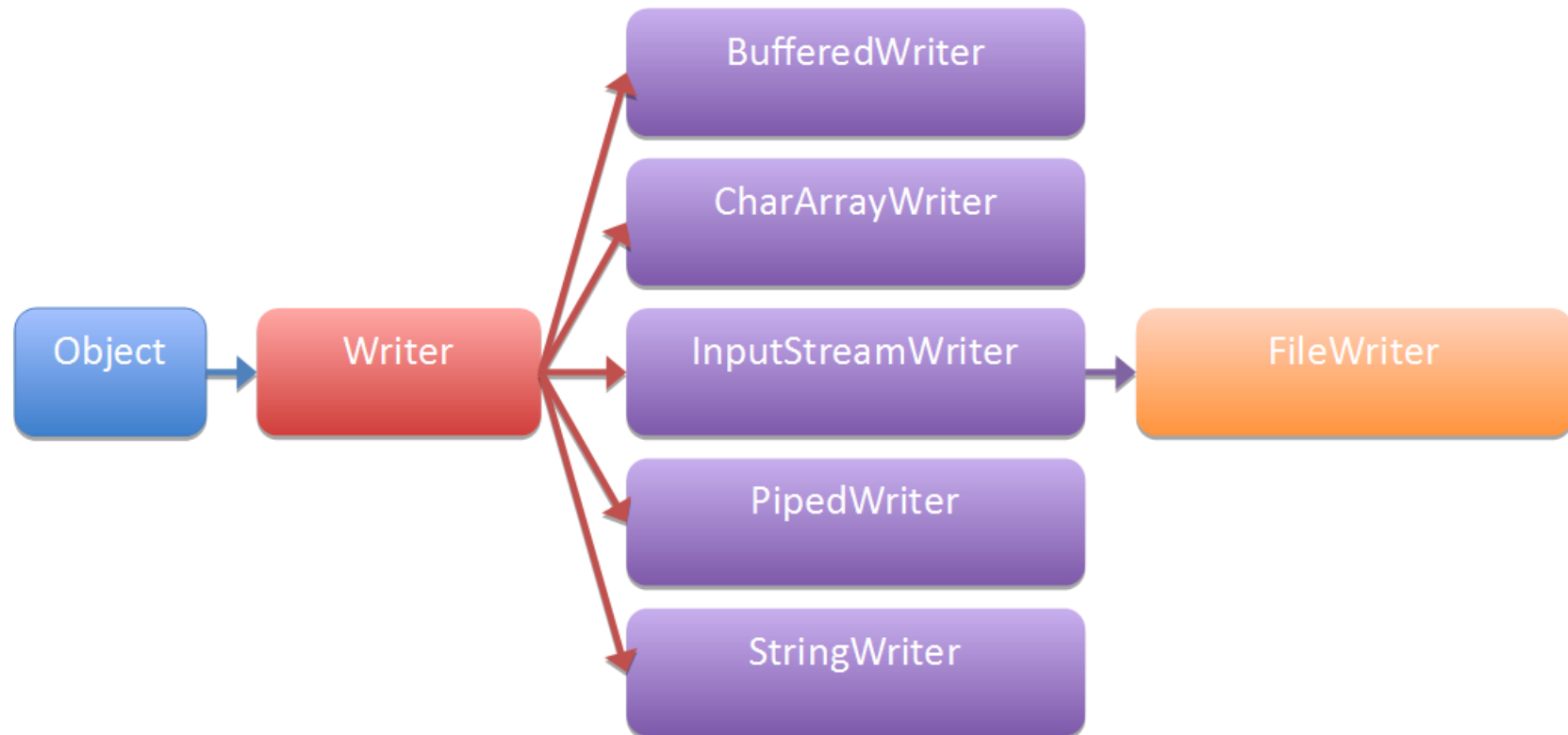
```
InputStream inputStream      = new FileInputStream("E:\\java.txt");
Reader    inputStreamReader = new InputStreamReader(inputStream);
int data = 0;
String StrConcatChar= new String();
while(data != -1)
{
    data = inputStreamReader.read();
    StrConcatChar += (char) data;
}
System.out.println(StrConcatChar);
inputStreamReader.close();
```



FLUX DE CARACTERE

LA CLASSE WRITER

Arborescence



Présentation : writer

Classe abstraite, super-classe de toutes les classes qui écrivent des flux de caractères.

Méthodes disponibles :

- `void write(char[] c)`
- `void write(char[] c, int off, int len)`
- `void flush()`
- `void close()`

Méthodes disponibles

La classes FileWriter écrit dans un fichier texte,

```
String StringBuffer = "Any String you want";  
  
//le file est crée si il n'existe pas  
File outputFile = new File("NoFileOnTheDisk.txt");  
FileWriter out = new FileWriter(outputFile);  
out.write(StringBuffer);  
out.close();
```

Ecriture en mode ajout:

```
File outputFile = new File("FileOnTheDisk.txt", true);
```

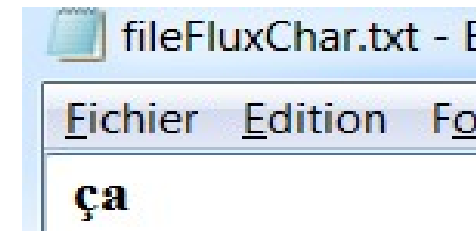
flux d'octets/de char

- les flux d'octets conviennent :
 - pour la copie conforme d'un contenu d'un fichier texte vers un autre sans autre opérations sur les caractères contenus dans le fichier
 - pour la manipulation d'image bmp (voire exo)
- Pas pour la lecture de fichier texte (contenant des caractères codés sur 2 octets)

flux d'octets/de char

Avec des caractères codés sur 2 octets (exemple : ç)
pour la lecture le choix d'un flux de char s'impose :

```
System.out.print("flux char lecture 2 octet par 2 octet: ");
Reader in = new FileReader("fileFluxChar.txt");
int data = 0;
while((data = in.read()) != -1)
    System.out.print((char)data);
in.close();
System.out.print("\n");
System.out.print("flux byte lecture octet par octet: ");
FileInputStream fis = new FileInputStream("fileFluxChar.txt");
while ((data = fis.read()) != -1)
    System.out.print((char)data);
fis.close();
```



```
flux char lecture 2 octet par 2 octet: ça
flux byte lecture octet par octet: Ãa
```

Buffer stream

Présentation

Lecture et écriture des données un octet à la fois
→ le programme devra accéder 1000 fois au disque pour lire un fichier de 1000 octets.

l'accès aux données sur le disque est bien plus lent que la manipulation de données en mémoire.

Pour minimiser le nombre de tentatives d'accès au disque, Java fournit ce qu'on appelle des tampons (buffers)

BufferedInputStream

La classe **BufferedInputStream** permet de remplir rapidement la mémoire tampon avec des données de **FileInputStream**.

Class name	Constructor's argument	Specificity
BufferedInputStream	InputStream is	Mise en tampon des octets lus à partir d'un InputStream

Temps de lecture du fichier :

```
FileInputStream fis = new FileInputStream("dictionnaire.txt");
byte[] buf = new byte[8];
long startTime = System.currentTimeMillis();
while(fis.read(buf) != -1);
fis.close();
System.out.println("temps:"+(System.currentTimeMillis() - startTime));
```

Le buffer stream charge en une fois dans un tampon en mémoire un paquet d'octets depuis un fichier.

/utilisation d'un buffer le compteur temp r.a.z

```
startTime = System.currentTimeMillis();
fis = new FileInputStream("dictionnaire.txt");
BufferedInputStream bis = new BufferedInputStream(fis);
```

Ensuite, la méthode read() lit chaque octet dans le tampon beaucoup plus rapidement qu'elle ne le ferait sur le disque.

```
while(bis.read(buf) != -1);
System.out.println("temps:"+(System.currentTimeMillis() - startTime));
```

temps en ms sans buffer:2122

temps en ms avec buffer:47

Des flux de buffer selon le stream

byte streams :

BufferedInputStream &
BufferedOutputStream

character streams :

BufferedReader & BufferedWriter

Utiliser flush pour vider le buffer.

Lire/crire le contenu du fichier en 1 seule fois

Methods for Small Files

Reading All Bytes or Lines from a File

Path file = ...;

byte[] fileArray;

fileArray = Files.readAllBytes(file);

Writing All Bytes or Lines to a File

Path file = ...;

byte[] buf = ...;

Files.write(file, buf);

Lecture partielle : `java.io.RandomAccessFile`

capability to move to different points in the file
and then read from or write to that location

Possibilité d'accès aléatoire du fichier

- Class supporting both file reading and writing
Mode Reading or Reading/Writing("r" or "rw")
- Use a movable pointer
void seek(long p): set the pointer to the position p
long getFilePointer(): returns the pointer's position
- Constructors:
`RandomAccessFile(File file, String mode)`
`RandomAccessFile(String filePath, String mode)`

RandomAccessFile: exemple

```
File myFile = new File("alphabet.txt");
RandomAccessFile raf = new RandomAccessFile(myFile, "r");
System.out.println("debut file: "+raf.getFilePointer());
raf.seek(24);
int data = 0;
while((data = raf.read()) != -1)
    System.out.println((char)data);
System.out.println(raf.getFilePointer());
raf.close();
```

Console

debut file: 0

y

z

26

Flux d'objet



Sérialisation

- Peut-on insérer un objet Java dans un fichier?
 - YES 😊
 - avec la **Sérialisation**
 - En devenant persistant, ils survivent à l'application.

“Serialization is the process of taking an object and converting it to a format in which it can be transported [...] to a storage location.”

– Mark Strawmyer

Outside Java with his little (very little) cousin Java Script
JSON makes it possible to store JavaScript objects as text.



Interface Serializable

- Si la classe implémente l'interface Serializable,
 - l'instance de classe devient persistant.
 - est sérialisable dans un flux

Exemple:

```
public class Person implements java.io.Serializable {  
    private String lastname, firstname;  
    private transient Date birthDate;  
}
```



Transient

Pour des problèmes de sécurité, certaines valeurs des variables ne doivent pas être sauvegardées.

→ Java fournit le modificateur **transient**

Les variables seront stockées (type et nom) mais pas leur valeur.

Une fois désérialisé, valeurs par défaut :

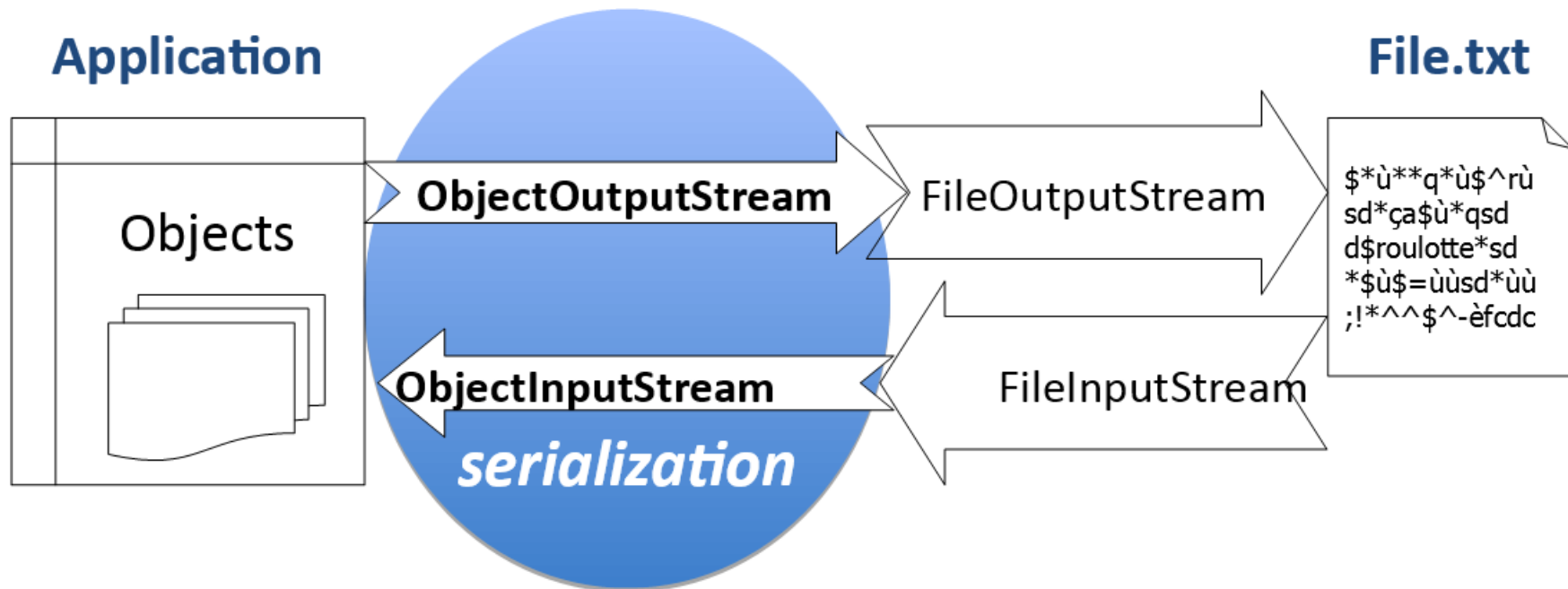
Number → 0

Boolean → false

Objet → Null

```
import java.io.Serializable;
public class MyClass implements Serializable {
    transient private String password;
}
```

Illustration





Classes & méthodes pour sérialiser les objets

- **ObjectOutputStream** – Sérialise l'objet
– **void writeObject(Object o)**
 - **ObjectInputStream** – De-serialize l'objet
– **Object readObject()**
 - Retourne un Objet, attention au cast 😊
- les données stockées sont les variables propres aux objets (son état interne)
- lors de la désérialisation, il est nécessaire de posséder la classe (le byte code) de l'objet à reconstruire.



Exemple – Classe sérialisable

```
public class Animal implements Serializable {  
  
    int age;  
    boolean vaccin;  
    String couleur;  
  
    public Animal(int i, boolean b, String str) {  
        this.age = i;  
        this.vaccin = b;  
        this.couleur = str;  
    }  
}
```



Exemple Serialization

```
FileOutputStream fos = null;
ObjectOutputStream oos = null;
try {
    Animal dog = new Animal(5, true, "black");

    fos = new FileOutputStream("SaveAnimal.txt");
    oos = new ObjectOutputStream(fos);

    oos.writeObject(dog); // Serialization
} catch (IOException e) {
    ...
} finally {
    // Close the streams
}
```

Animal
-age:int
-vaccination:Boolean
-color:String



Exemple – De-Serialization

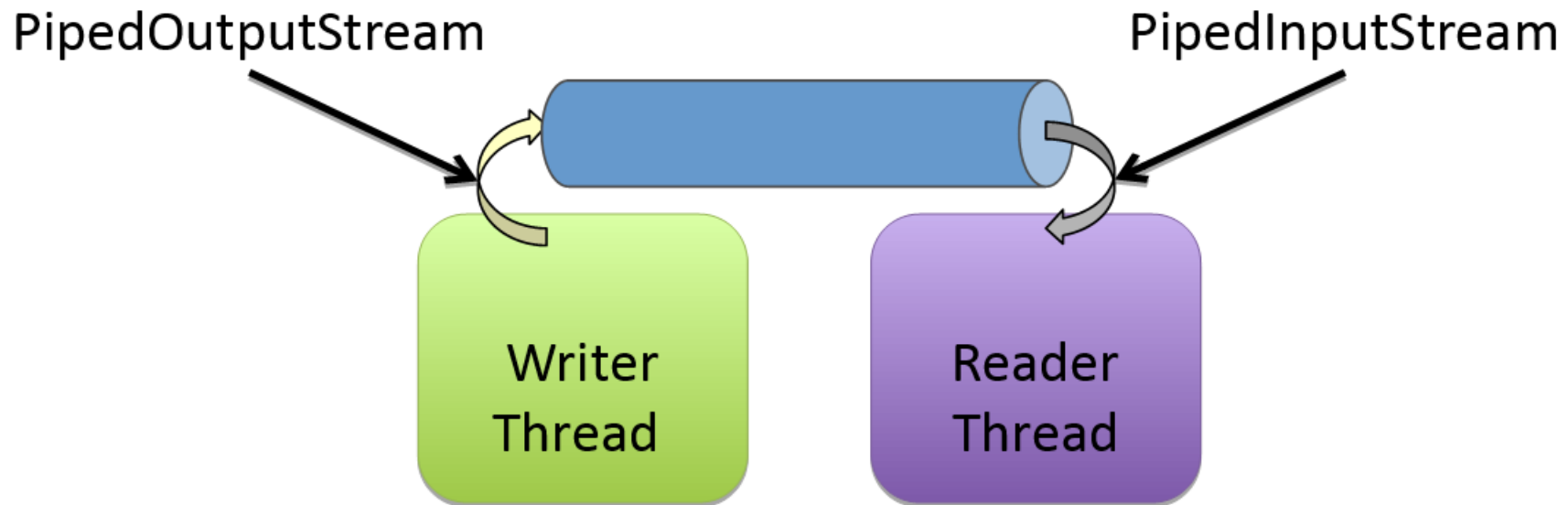
```
FileInputStream fis = null;
ObjectInputStream ois = null;
try {
    fis = new FileInputStream("SaveAnimal.txt");
    ois = new ObjectInputStream(fis);
    Animal dog = (Animal) ois.readObject();
    System.out.println("Age of my dog: " + dog.getAge());
    System.out.println("Color of my dog: " + dog.getColor());
    if (dog.isVaccinated())
        System.out.println("My animal is vaccinated");
    } catch (IOException e) {
        ...
    } finally { /* Close the stream */ }
```



Piped streams



Illustration





Objet pour créer un tube entre 2 threads

- **PipedInputStream** (or **PipedReader**)

- Must be connected to a *PipedOutputStream*
- Read the data from the pipe (acts as a Reader Thread)

– **PipedOutputStream** (or **PipedWriter**)

- Write data in the pipe (acts as a Writer Thread)

- close your Piped Stream in your Thread's runnable after writing and reading bytes to/from the pipe stream.

Exemple

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

    PipedOutputStream output = new PipedOutputStream();
    PipedInputStream input = new PipedInputStream();
    input.connect(output);

    class WriterThread extends Thread {
        //redefinition méthode run de la classe thread
        public void run() {
            output.write("Hello world par le pipe!".getBytes());
            output.close();
        }
    }

    class ReaderThread extends Thread {
        public void run(){
            int data ;
            while(( data = input.read())!= -1)
                System.out.print((char) data);
            input.close();
        }
    }

    new WriterThread().start();
    new ReaderThread().start();

}
```


TRAVAILLER AVEC LE FORMAT ZIP

- compression stream**
- décompression stream**



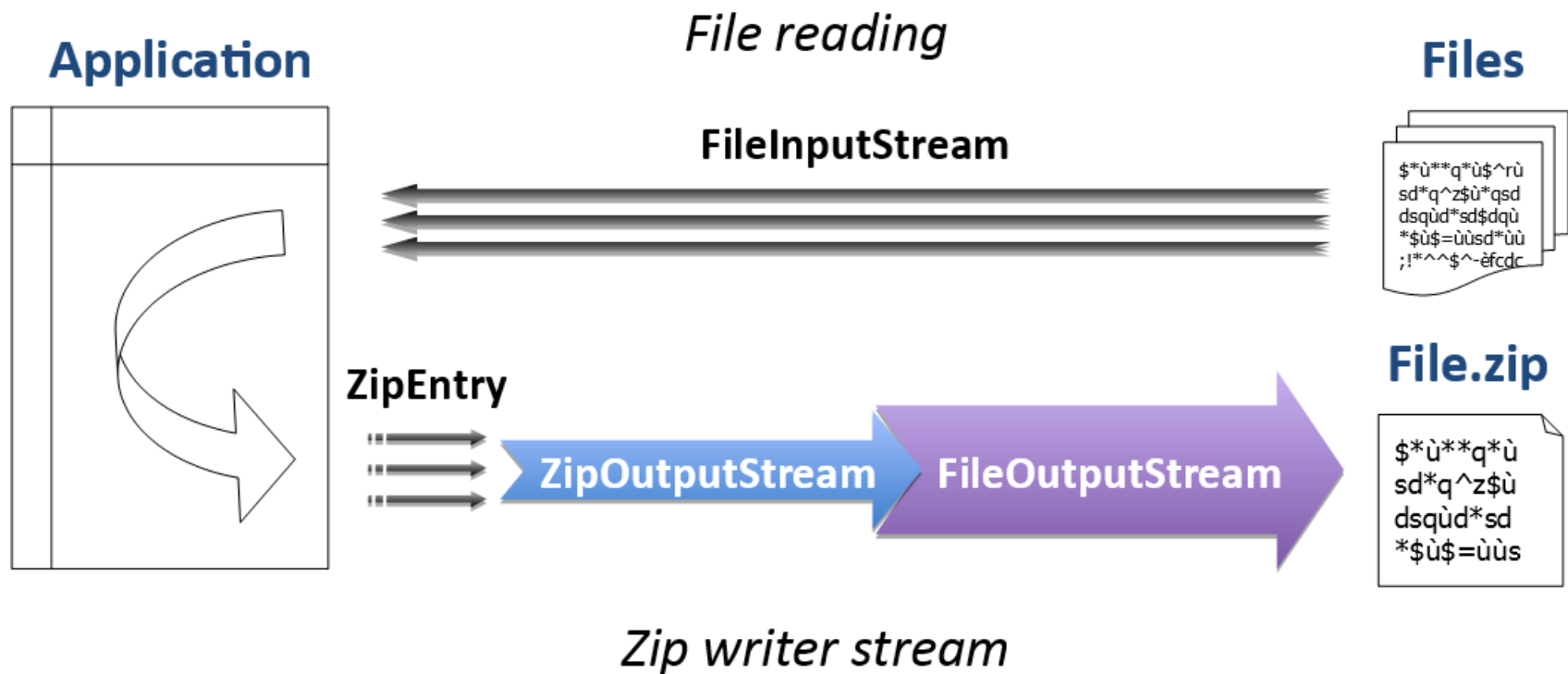
ZipOutputStream/ZipEntry (java.util.zip)

Un objet de la classe ZipEntry représente une entrée dans le fichier zip (constructeur prend en param le nom du fichier)

- A partir de l'objet appel des méthodes :
 - `void putNextEntry(ZipEntry ze)`
- toutes données sont associées à ce flux jusqu'à l'appel de
 - `void closeEntry()`
- écriture → méthode write:
 - `void write(byte[] b)`
 - `void write(byte[] b, int off, long len)`



Schema





zip

compression stream: par étapes

```
//declaration des variables
try
{
    //1) instancier un FileOutputStream sur le fichier zip
    fos = new FileOutputStream("archivedico.zip");
    //2) instancier un ZipOutputStream sur l'objet FileOutputStream
    zipos = new ZipOutputStream(fos);
    byte[] data = new byte[2048];
    fis = new FileInputStream("dico.txt");
    //3) créer un Zip entry pour le fichier à compresser
    ze = new ZipEntry("dico.txt");
    //4) le fichier est placé dans le zip
    zipos.putNextEntry(ze);
    //5) écriture des données dans l'outputstream zip
    while (fis.read(data) != -1) {zipos.write(data);}
    //6) fermeture des flux
    zipos.closeEntry();zipos.close();
}
catch (Exception e){}
```



dico.txt

Document texte

3 976 Ko



archivedico.zip

Dossier compressé

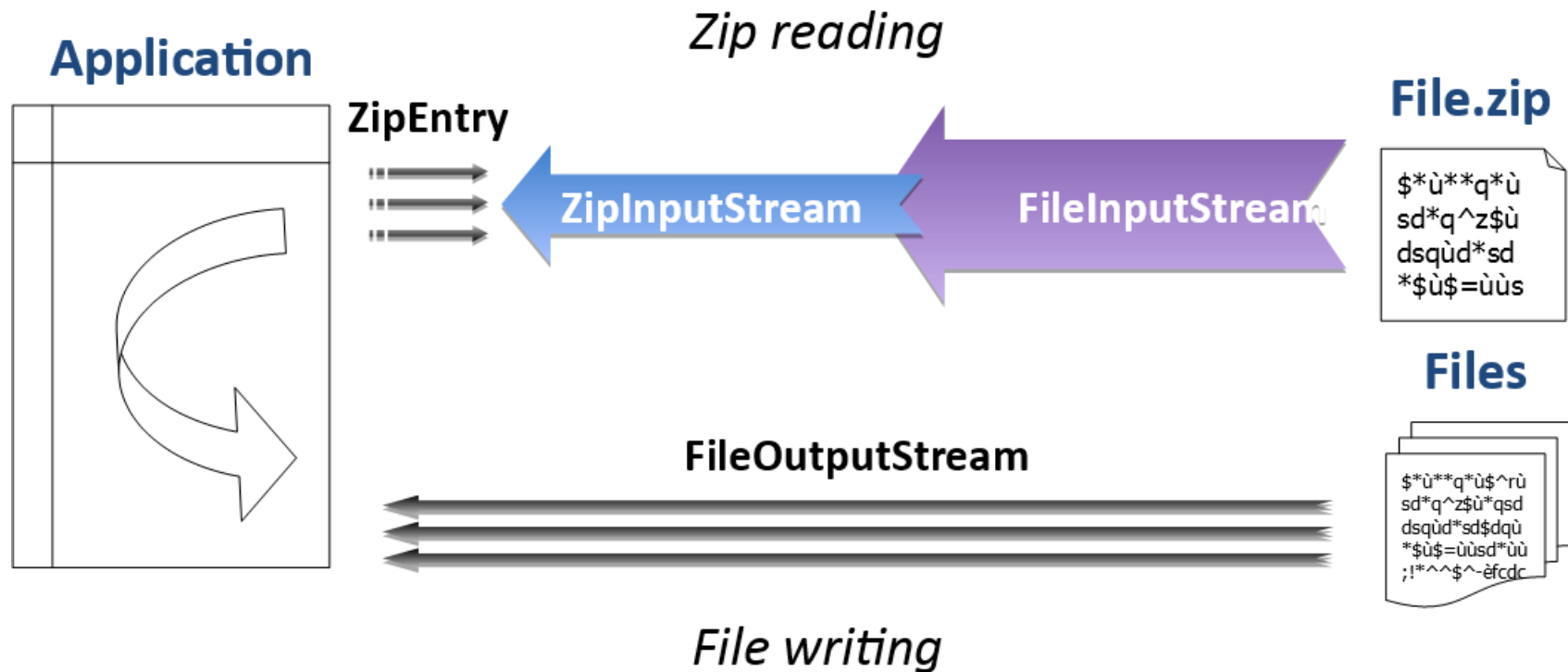
807 Ko



Decompression stream

- ZipInputStream useful methods:
 - `int read(byte[] b)`
 - `int read(byte[] b, int off, long len)`
 - `ZipEntry getNextEntry()`

Decompression - Schema





zip

Decompression Stream – par étapes

```
try {  
    //1) instancier un FileInputStream sur le fichier zip  
    fis = new FileInputStream("archivedico.zip");  
    //2) instancier un ZipInputStream sur l'objet FileInputStream  
    zis = new ZipInputStream(fis);  
    //3) récupérer le zipentry pour déterminer le fichier d'origine  
    while ((ze = zis.getNextEntry()) != null) {  
        int b;  
        //4) récupérer le nom du fichier d'origine & un flux  
  
        fos = new FileOutputStream(ze.getName());  
        while ((b = zis.read()) != -1) {  
            fos.write(b); //5) écriture dans ce fichier  
        }  
        fos.close();  
    }  
} catch (Exception e) { /* ... */ }  
finally { /* ... */ }
```



Quiz

Difference between InputStream & Reader/Writer ?

InputStream = bytes streams

Reader/Writer = chars streams

What to do after having used a stream ?

Close it

Is File an abstract class ?

No it's an abstract representation



Quiz

Question 1. What class and method would you use to read a few pieces of data that are at known positions near the end of a large file?

Answer 1. `Files.newByteChannel` returns an instance of `SeekableByteChannel` which allows you to read from (or write to) any position in the file.

What is the effect of transient ?

The property won't be serialized

Which interface must be implemented to serialize a class ?

Serializable



Exercice (1/4)

- We're going to develop a simple application for file compression and decompression.
 - A sort of gzip with less functionalities and in Java !

```
brice-argensons-macbook:~ derf4002$ java -jar jzipper.jar
Usage: jzipper [OPTION] [FILE]...
Compress or uncompress FILES.

Option can be:
    compress [ARCHIVE_NAME]    Compress one or several files inside a new archive.
    decompress                 Decompress files inside a archive.

brice-argensons-macbook:~ derf4002$
```



Exercise (2/4)

- Three options must be available :
 - **compress [archiveName] [file]...:**
 - To compress the specified files inside an archive with the specified name.
 - **decompress [archiveName] :**
 - To decompress the specified archive in the current folder.
 - **help:**
 - To display some information about how to use the application.



Exercise (3/4)

```
Terminal — bash — 93x13
bash      vim      bash
brice-argensons-macbook:~ derf4002$ java -jar jzipper.jar
Usage: jzipper [OPTION] [FILE]...
Compress or uncompress FILES.

Option can be:
    compress [ARCHIVE_NAME]    Compress one or several files inside a new archive.
    decompress                 Decompress files inside a archive.

brice-argensons-macbook:~ derf4002$ java -jar jzipper.jar compress Archive.zip bitmap_new.bmp
bitmap.bmp
brice-argensons-macbook:~ derf4002$ java -jar jzipper.jar decompress Archive.zip
brice-argensons-macbook:~ derf4002$
```



Exercise (4/4)

- It must be composed of at least three classes :
 - **Launcher:**
 - Containing the main methods which must manage the arguments passed by the user.
 - **ZipCompressor:**
 - Containing the code to compress files inside an archive.
 - **ZipDecompressor:**
 - Containing the code to decompress files inside an archive.

INPUT OUTPUT

