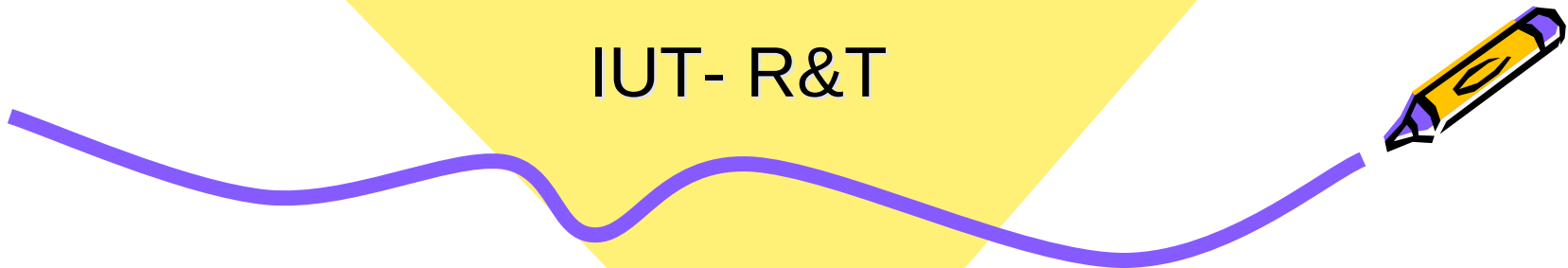




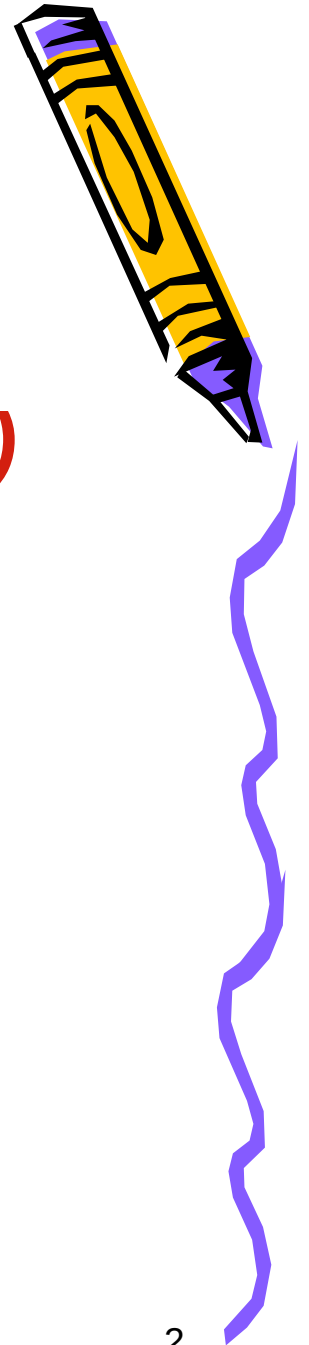
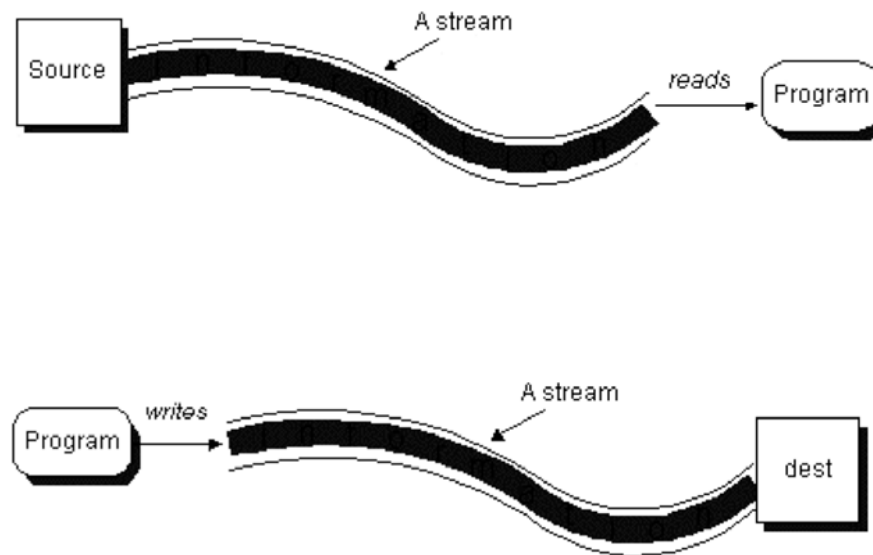
Entrées/Sorties

Karima Boudaoud
IUT- R&T

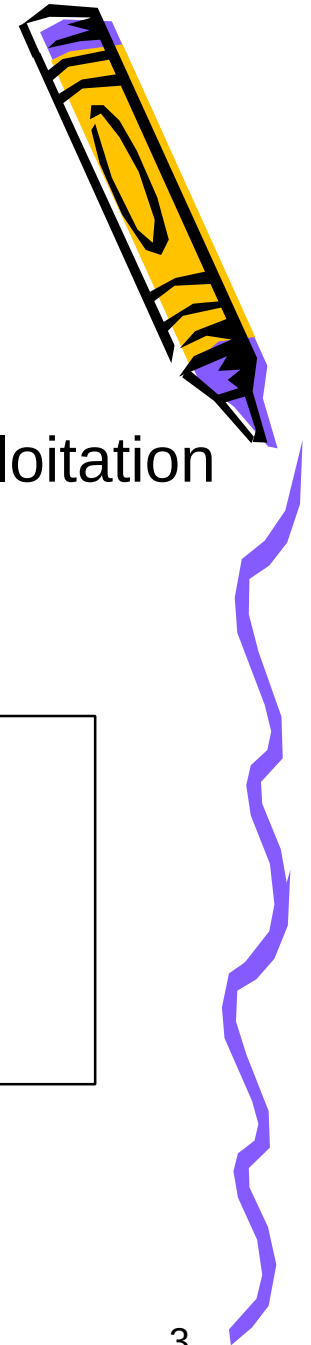


Entrées / Sorties

○ Fonctionnement par flots (*stream*)



Entrées / Sorties



○ Terrain connu

- la classe `java.lang.System`
- cette classe gère l'interface avec le système d'exploitation
`System.out.println("Accrochez-vous !");`

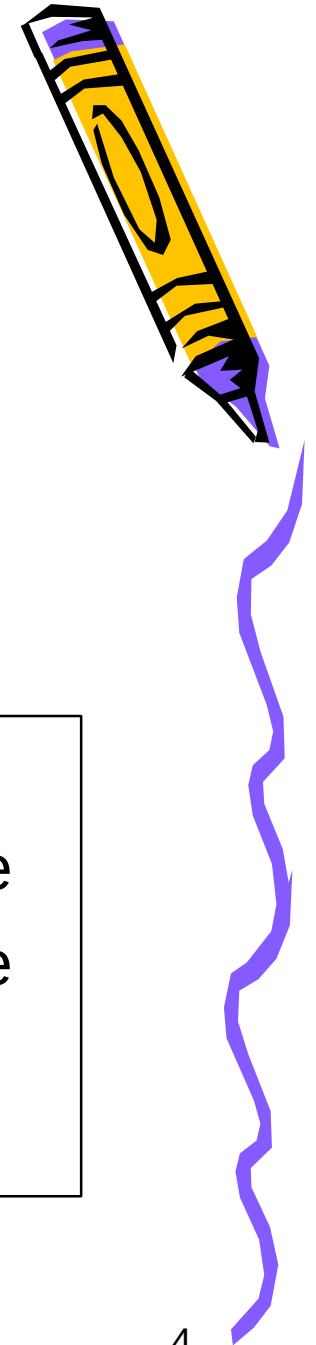
○ 3 flots standards

```
System.in;    // entrée à la console  
System.out;   // affichage à l'écran  
System.err;   // affichage à l'écran
```

- ce sont des objets **static**



System.out



○ Type

- c'est un objet de type `java.io.PrintStream`

○ Caractéristiques

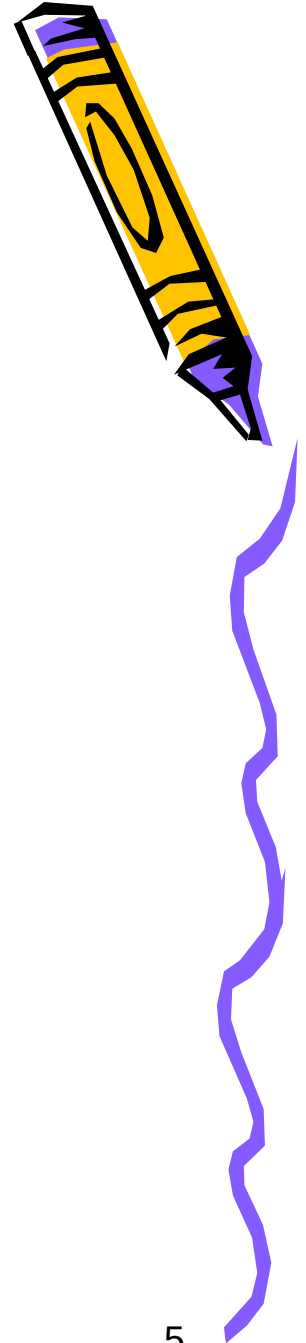
- produit un flot d'octets formaté pour affichage
- possède une méthode **print** pour chaque type

```
print(int i); // octets pour entier  
print(double d); // octets pour double  
print(char c); // octets pour caractère  
...  
print(Object o); // o.toString()
```



System.err

- idem que **System.out**
- permet d'indiquer des erreurs



System.in

○Type

- c'est un objet de type `java.io.InputStream`

○Caractéristiques

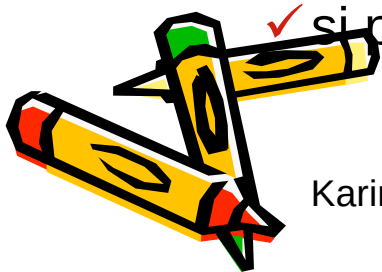
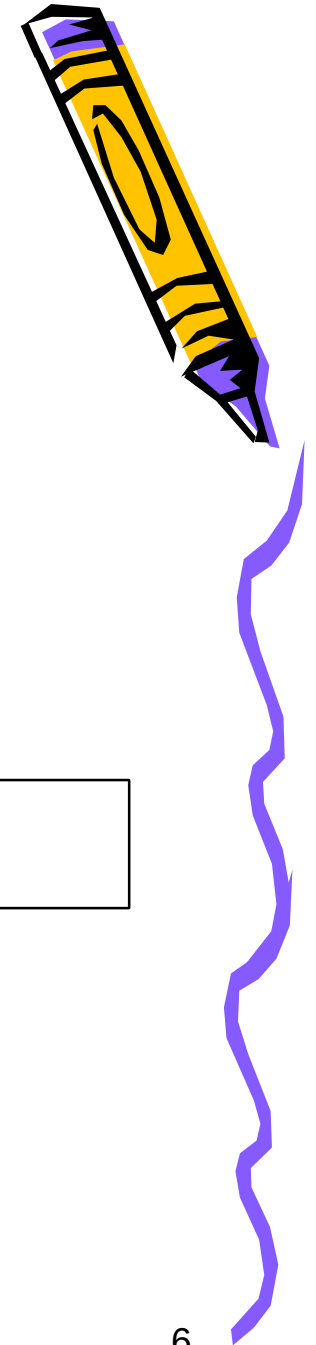
- comme pour **OutputStream**, il propose une fonctionnalité de base
- possède une seule méthode

```
public int read(); // lit un octet
```

✓ attend le prochain octet en entrée

▲ retourne $0 < \text{son code} < 255$

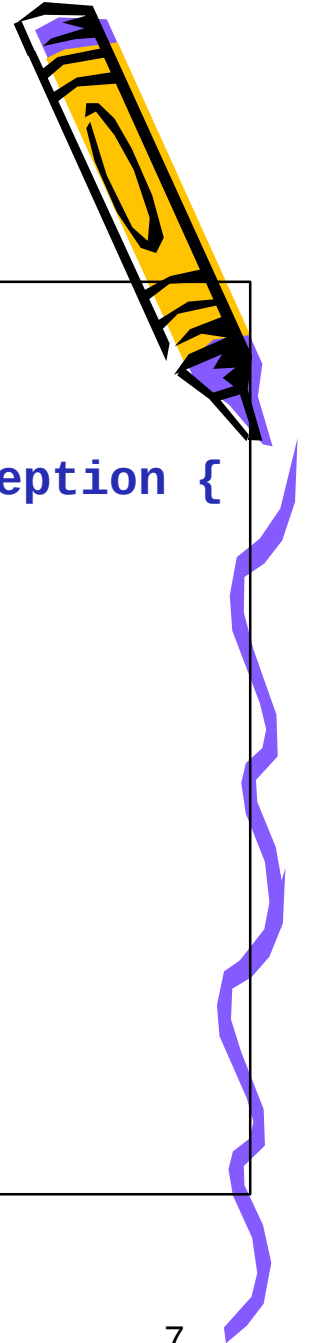
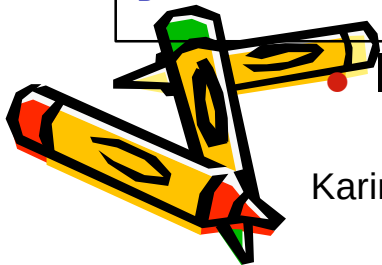
✓ si plus rien, renvoie -1



Exemple

```
import java.io.*;
public class Naif {
public static void main(String[] args) throws IOException {
    int b, nb = 0;
    // lecture de caractères au clavier
    while((b = System.in.read()) != -1) {
        nb++;
        System.out.print((char) b);
    }
    System.out.println("nb = " + nb);
}
```

• le résultat peut surprendre...



Paquetage java.io

Object

InputStream

Lecture d'octets

OutputStream

Ecriture d'octets

Reader

Lecture de caractères Unicode

Writer

Ecriture de caractères Unicode

File

Maniement de noms de fichiers
et de répertoires

StreamTokenizer

Analyse lexicale d'un flot d'entrée

Lecture et écriture de flots d'octets

Lecture et écriture de flots de
caractères *Unicode*

Pour représenter fichiers et répertoires

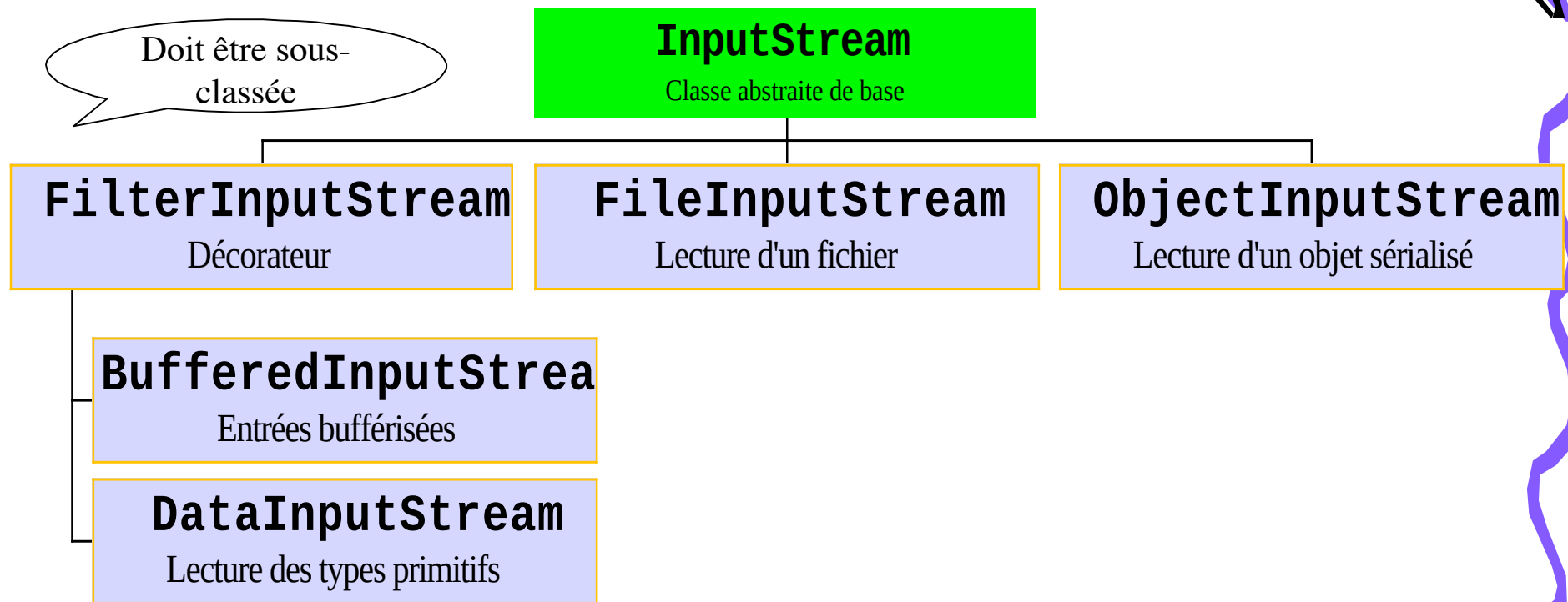
Analyse lexicale d'un flot d'entrée



Lecture de flots d'octets

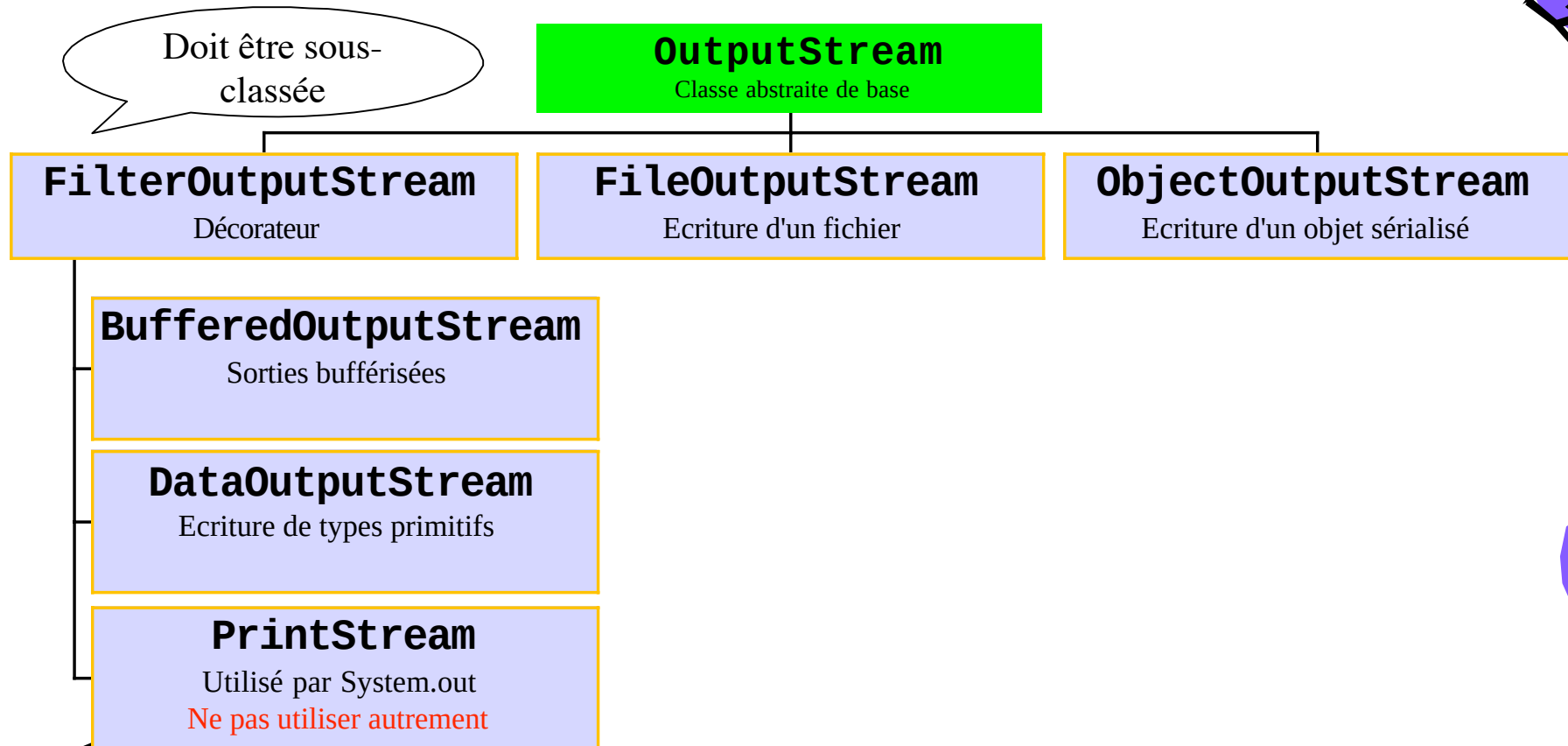


- Hiérarchie des principales classes de lecture d'un flot d'octets



Écriture de flots d'octets

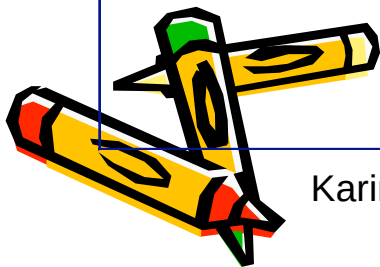
- Hiérarchie des principales classes d'écriture d'un flot d'octets



Lecture et écriture de flots d'octets

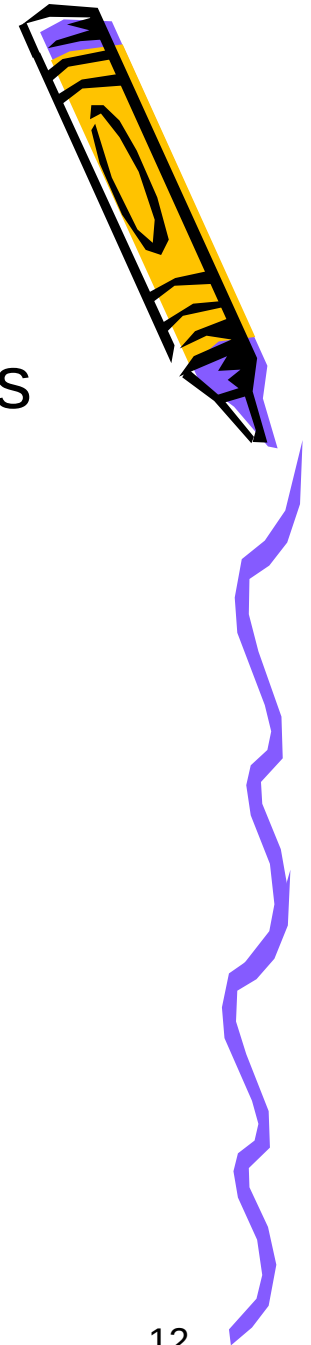


Classe pour entrée	Classe pour sortie	Fonctions fournies
InputStream	OutputStream	Classes abstraites de base pour lecture et écriture d'un flot de données
FilterInputStream	FilterOutputStream	Classe mère des classes qui ajoutent fonctionnalités à Input/OutputStream
BufferedInputStream	BufferedOutputStream	Lecture et écriture avec buffer
DataInputStream	DataOutputStream	Lecture et écriture des types primitifs
FileInputStream	FileOutputStream	Lecture et écriture d'un fichier
	PrintStream	Possède les méthodes print() , println() utilisées par System.out



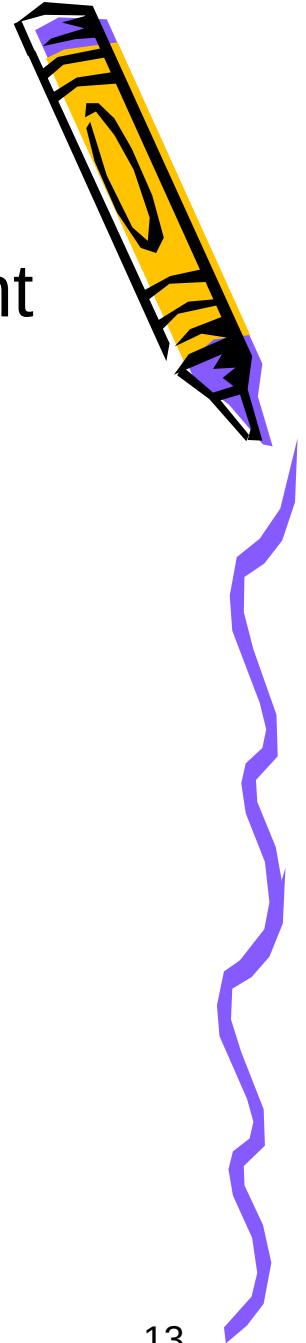
Package java.io

- Le package **java.io** fournit des classes permettant de manipuler diverses ressources
 - ✓ fichiers
 - ✓ mémoire
 - ✓ tuyaux (*pipes*)
 - ✓ filtres

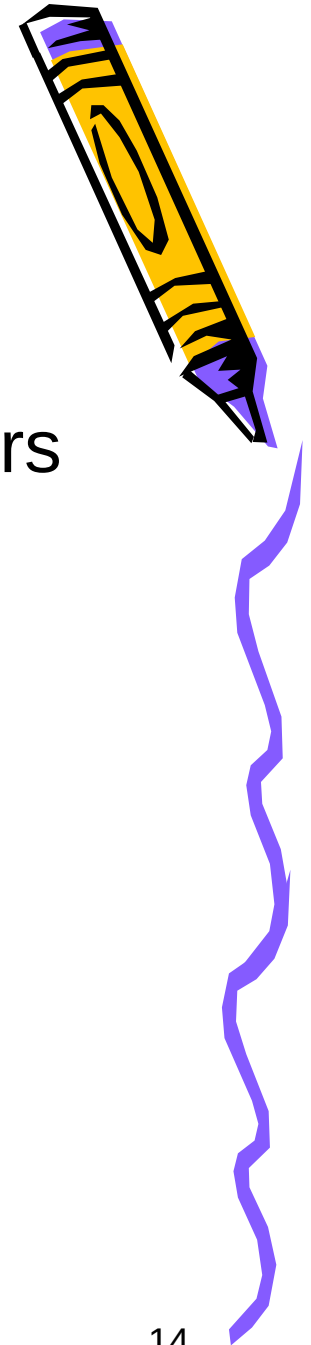


Les Fichiers

- les classes manipulant les fichiers se trouvent dans la classe **java.io.File**
- il existe trois classes principales
 - ✓ **FileInputStream**
 - ✓ **FileOutputStream**
 - ✓ **RandomAccessFile**



java.io.File (1)



- la classe **java.io.File** fournit de nombreuses méthodes pour gérer des fichiers et répertoires
 - ✓ `String getName()`
 - ✓ `String getPath()`
 - ✓ `String getAbsolutePath()`
 - ✓ `String getParent()`
 - ✓ `boolean renameTo(File newName)`



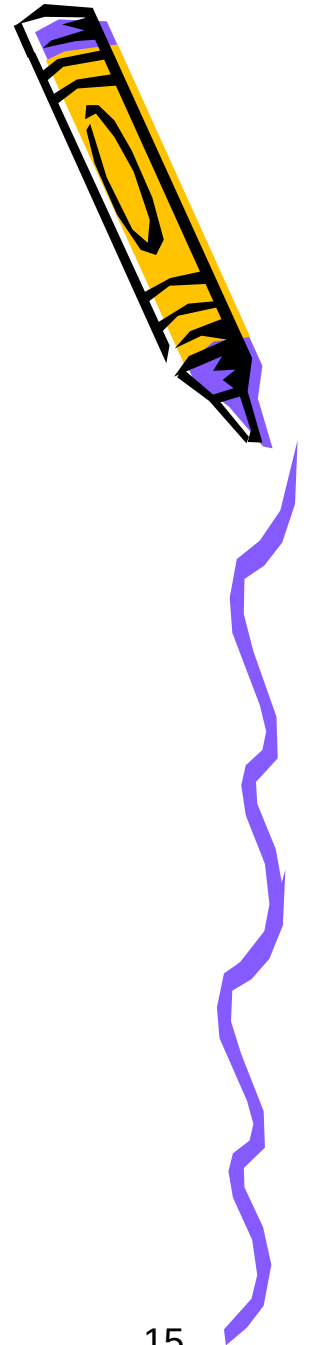
java.io.File (2)

○ Droits d'accès

- ✓ `boolean exists()`
- ✓ `boolean canWrite()`
- ✓ `boolean canRead()`
- ✓ ...

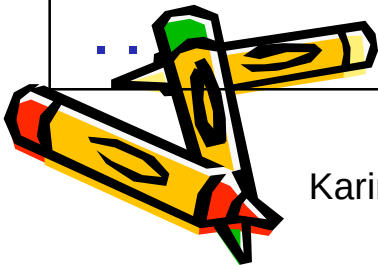
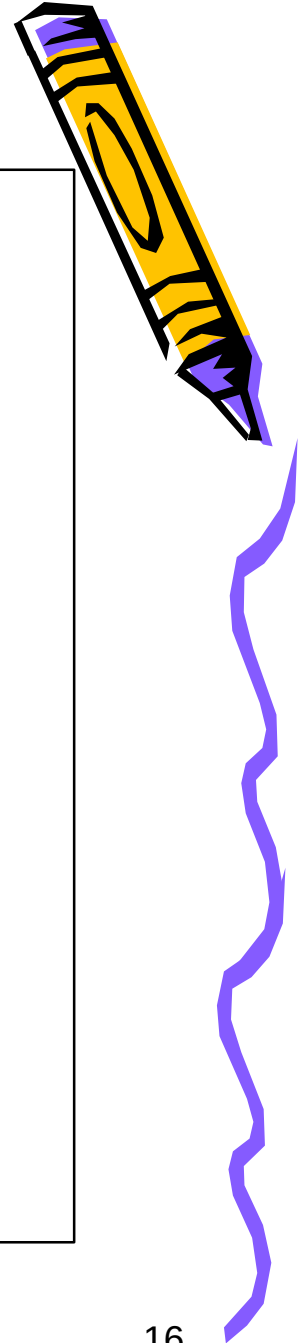
○ Autres informations

- ✓ `long length()`
- ✓ `long lastModified()`
- ✓ `boolean delete()`

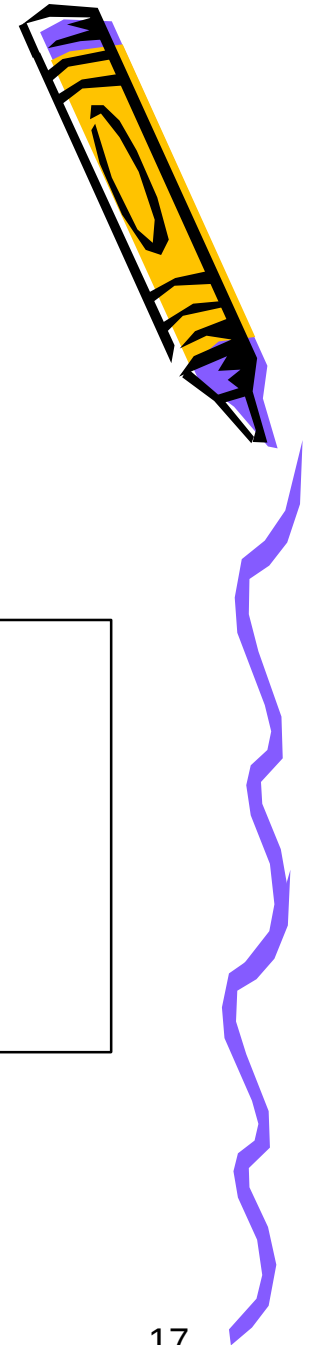


java.io.File (3)

```
...  
File f = new File("toto.txt");  
System.out.println("toto.txt : "  
    + f.getName() + " " + f.getPath());  
if (f.exists()) {  
    System.out.println(  
        "Fichier existe, droits = "  
        + f.canRead() + " " + f.canWrite()  
        + " " + f.length());  
    ...  
}
```



java.io.File (4)



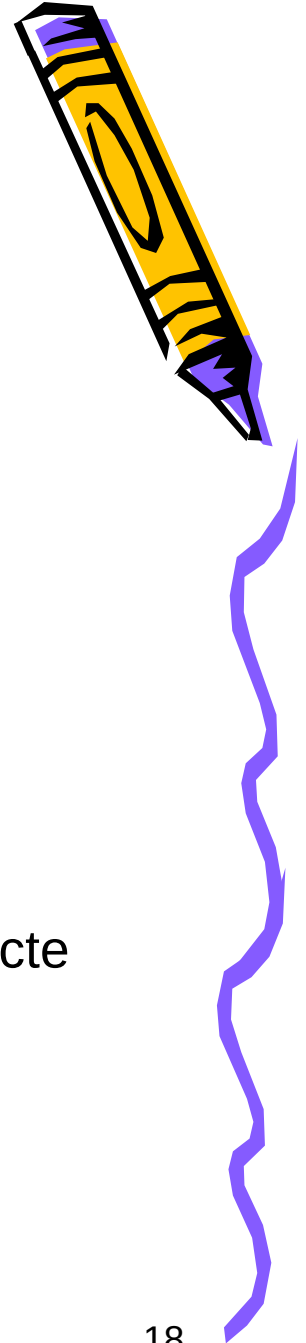
○ Répertoires

- ✓ `boolean mkdir()`
- ✓ `String[] list()`

```
File f = new File("/u/I3S/buffa", ".emacs");  
// ou bien  
File home = new File("/u/I3S/buffa");  
File f = new File(home, ".emacs");
```



Package java.io (1)



○Mémoire

- ✓ **BufferedInputStream**
- ✓ **BufferedOutputStream**

▲ pour les lectures bufférisées

- ✓ **DataInputStream**

- ✓ **DataOutputStream**

▲ lectures typées

▲ utile pour lire des fichiers dont on connaît la nature exacte

– 3 doubles et un entier sur chaque ligne par exemple



Package java.io (2)



○Tuyau

- ✓ PipedInputStream
- ✓ PipedOutputStream

- ces 2 classes permettent à deux *threads* d'échanger des données.

○Lecture filtrée

- ✓ StreamTokenizer



java.io.InputStream (1)

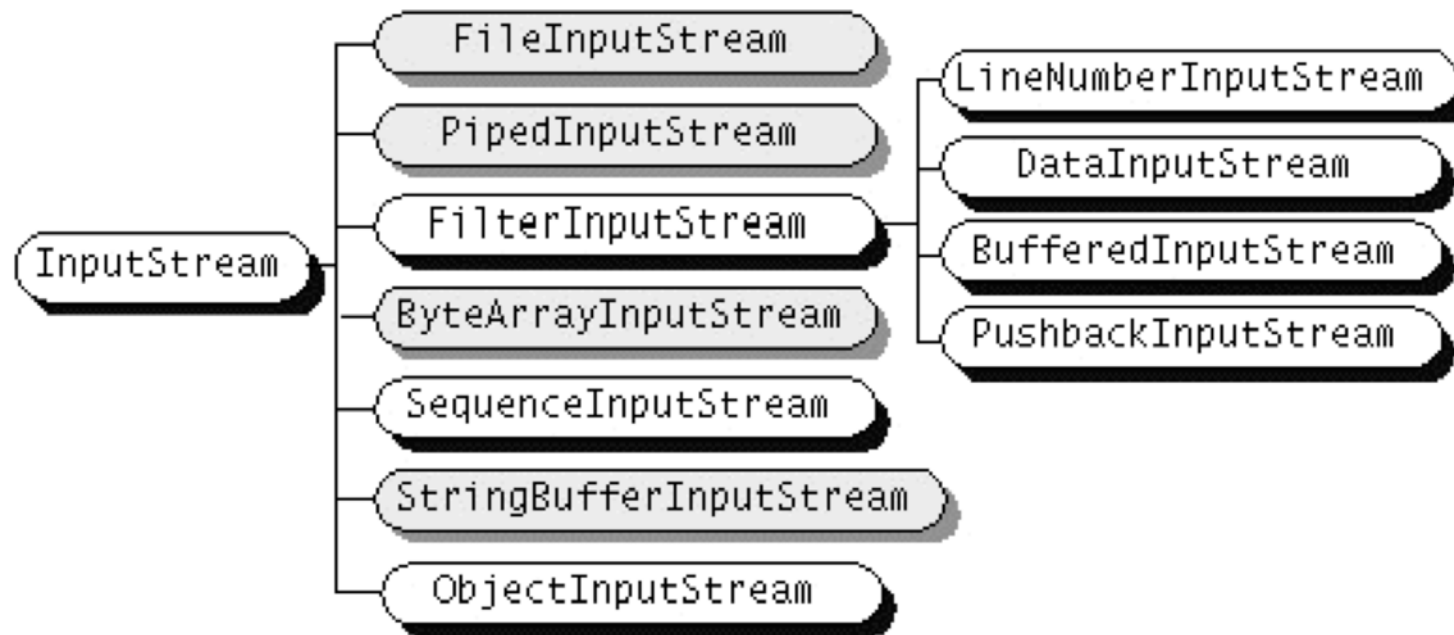


○Caractéristiques

- la classe **InputStream** est une classe abstraite
- elle oblige les sous-classes à implémenter la méthode **read**
 - ✓ Lecture d'octets uniquement
- c'est la classe de base de tous les flots d'entrée



java.io.InputStream (2)



java.io.InputStream (3)

- Infos sur quelques sous-classes

- ✓ **SequenceInputStream**

- permet de travailler avec une énumération d'**InputStreams** comme s'il s'agissait d'un seul

- ✓ **StringBufferInputStream**

- lecture de chaînes de caractères

- ✓ **ByteArrayInputStream**

- lecture de tableaux d'octets

- ✓ **FilterInputStream**

- encapsule une instance d'**InputStream** et fournit des méthodes de filtrage



java.io.FileInputStream (1)

○Lecture

- cette classe permet de lire des caractères ou des octets d'un fichier

○Fermeture

- la fermeture d'un fichier se fait par la méthode **close**

○Constructeurs

```
public FileInputStream(String filename)  
public FileInputStream(File file)
```



java.io.FileInputStream (2)



○ Méthodes de lecture

- toutes les méthodes renvoient **-1** en fin de fichier

✓ **int read()**

- lit un octet

✓ **int read(byte[] taboctets)**

- remplit **taboctets** si possible
- **taboctets** doit être alloué avant

✓ **int read(byte[] taboctets, int offset, int nb)**

- lit nb octets et les met dans **taboctets** à partir de l'indice **offset**.



java.io.FileInputStream (3)

```
FileInputStream fis;  
byte[] b = new byte[1024];  
try { // ouverture du fichier  
    fis = new FileInputStream("/u/I3S/buffa/.emacs");  
} catch (FileNotFoundException e) {...}  
try { // lecture des données  
    int i = fis.read(b);  
} catch (IOException e) {...}  
// utilisation des caractères lus  
String s = new String(b, 0);
```



java.io.DataInputStream (1)



○Lecture

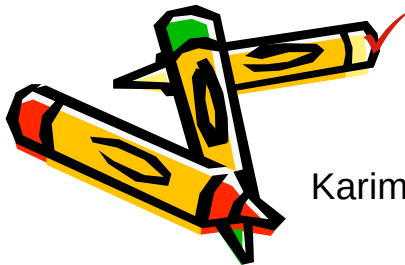
- cette classe permet la lecture de données typées
- en terme de portabilité, un entier est de même taille partout

○Classe de "spécialisation"

- cette classe peut transformer en **DataInputStream** n'importe quel **InputStream**

○Constructeur

public DataInputStream(InputStream)



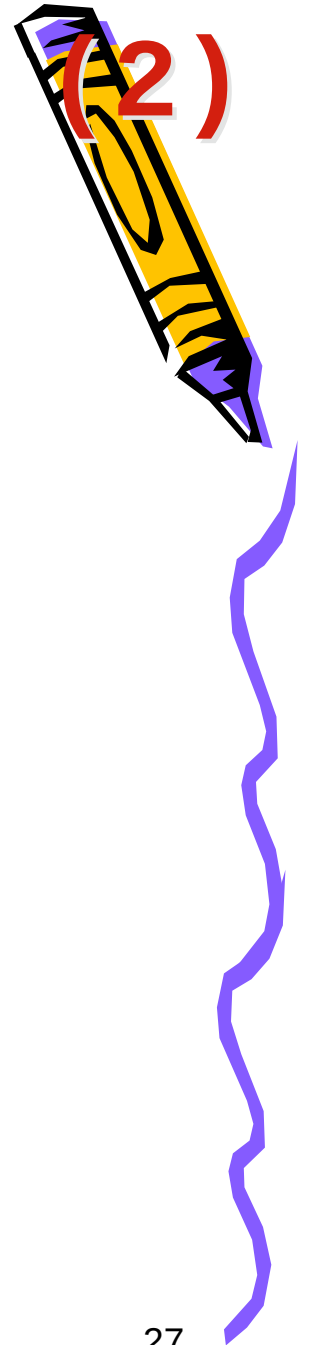
java.io.DataInputStream (2)

○ Méthodes de lecture

- ✓ `byte readByte()`
- ✓ `short readShort()`
- ✓ `char readChar()`
- ✓ `int readInt()`
- ✓ `float readFloat()`
- ✓ `String readLine`

▲ la fin de la ligne marquée par `\n`, `\r`, `\r\n`, **EOF**

✓ ...



java.io.DataInputStream (3)

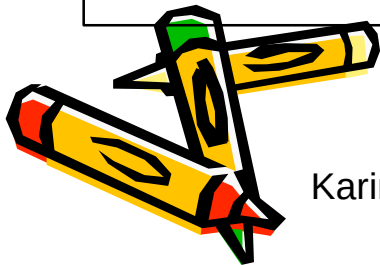
○ Fermeture de flot

- la méthode qui permet de fermer un flot est la méthode **close**
- la fermeture de entraîne la fermeture de **InputStream**



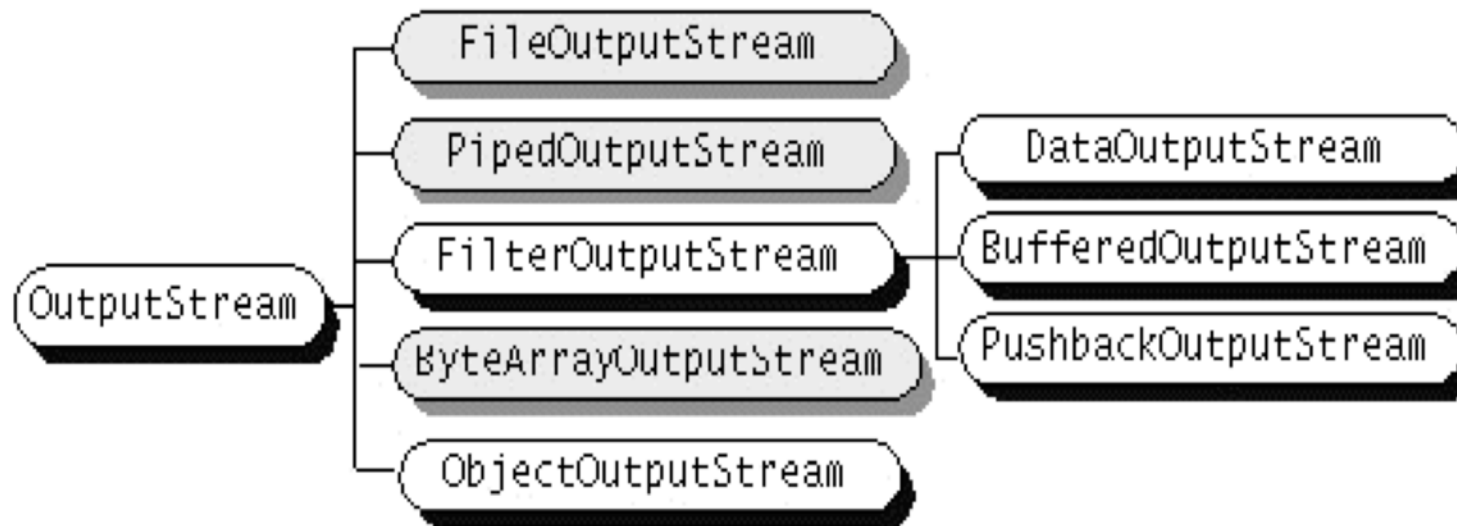
java.io.DataInputStream (4)

```
// Ouverture d'un fichier
FileInputStream fis;
fis = new FileInputStream("ball.obj");
// spécialisation !
DataInputStream dis = new DataInputStream(fis);
String ligne = new String();
// lecture des données
while((ligne = dis.readLine()) != null){
    System.out.println(ligne + "\n");
}
dis.close(); // fermeture du fichier. Ferme aussi fis !
```



java.io.OutputStream (1)

- la classe **OutputStream** est une classe abstraite
- elle est la de base de tous les flots de sortie



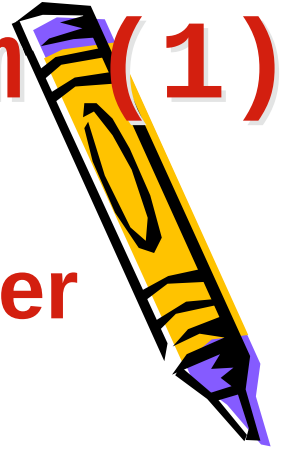
java.io.OutputStream (2)

○ La classe `PrintStream(OutputStream os)`

- la classe `PrintStream` permet d'écrire des données typées autres que des **bytes**
- la classe **`System.out`** est un **`PrintStream`**
- les méthodes fournies par cette classe sont
 - ✓ `close()`
 - ✓ `flush()`
 - ✓ `write()` // permet l'écriture d'octets
- ces méthodes sont implémentées par les sous-classes de **`OutputStream`**



java.io.FileOutputStream(1)

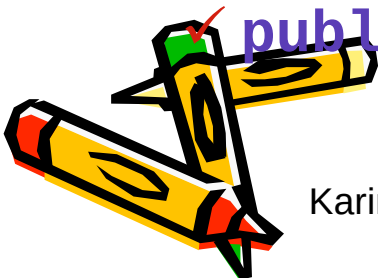


○Écriture de caractères dans un fichier

- l'ouverture du fichier est faite à la construction
 - ✓ Le **SecurityManager** est appelée à chaque ouverture de fichier
- il existe des méthodes d'écriture d'octets.
- la fermeture est soit explicite avec **close()** ou implicite

○Constructeurs

- ✓ **public FileOutputStream(File)**
- ✓ **public FileOutputStream(FileDescriptor)**
- ✓ **public FileOutputStream(String)**



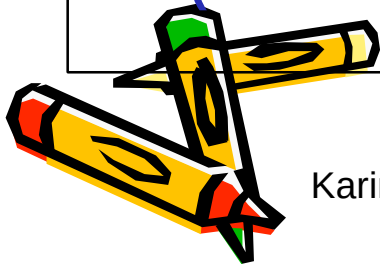
java.io.FileOutputStream (2)

○ Méthodes d'écriture

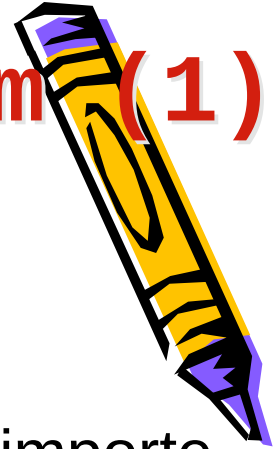
✓ `public void write(int)`

✓ `public void write(byte [])`

```
FileOutputStream fos =  
    new FileOutputStream("toto");  
String chaine = new String("Coucou c'est moi");  
int longueur = chaine.length();  
byte[] buffer = new byte[longueur];  
chaine.getBytes(0, longueur - 1, buffer, 0);  
for(int i = 0; i < longueur; i++)  
    fos.write(buffer[i]);
```



java.io.BufferedOutputStream (1)



○Écriture

- l'écriture se fait sur un flot de sortie bufférisé
- la spécialisation des données se fait à partir de n'importe quel flot de sortie.
- l'écriture sur disque se fait de manière explicite par la méthode **flush()** ou la méthode **close()**

○Constructeurs

- ✓ **BufferedInputStream(OutputStream)**
- ✓ **BufferedOutputStream(OutputStream, int size)**



java.io.BufferedOutputStream (2)

○ Méthodes

- ✓ `write(int)`
- ✓ `write(byte [], int offset, int length)`
- ✓ `flush()`
- ✓ `close()`

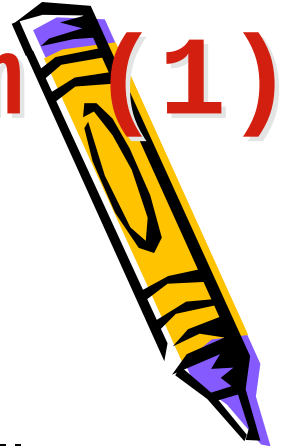


java.io.BufferedOutputStream (3)

```
// ouverture d'un fichier pour écriture en mode bufférisé
FileOutputStream fos = new FileOutputStream("toto.txt");
BufferedOutputStream bos = new BufferedOutputStream(fos);
// préparation des données à écrire
String chaine = "coucou c'est moi";
int longueur = chaine.length();
byte[] buffer = new byte[longueur];
chaine.getBytes(0, longueur - 1, buffer, 0);
// écriture
for(int i = 0; i < longueur; i++) {
    bos.write(buffer[i]);
}
bos.write("\n");
bos.close();
```



java.io.DataOutputStream (1)



○Écriture

- cette classe permet l'écriture de données typées
- en terme de portabilité, un entier est de même taille partout

○Classe de "spécialisation"

- cette classe permet de transformer en **DataOutputStream** n'importe quel **OutputStream**
- il est conseillé de prendre un **BufferedOutputStream**

○Constructeur

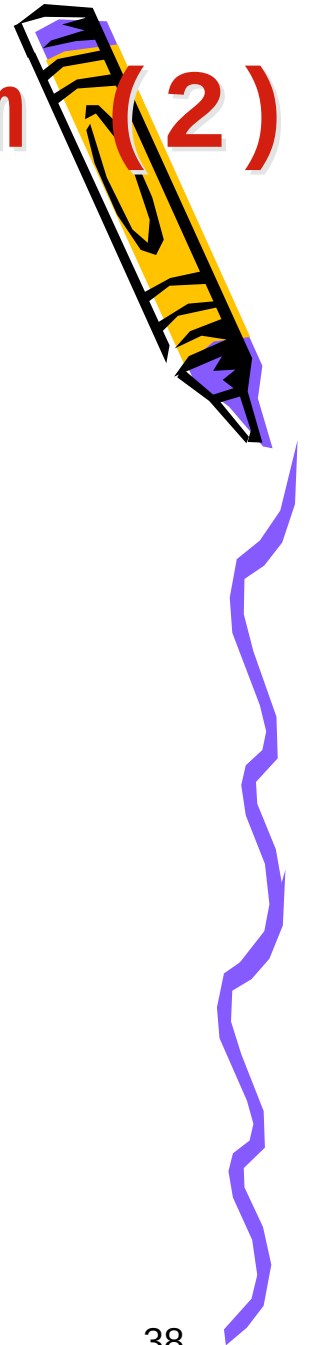
✓ `public DataOutputStream(OutputStream)`



java.io.DataOutputStream (2)

○ Méthodes d'écriture

- ✓ `byte writeByte()`
- ✓ `short writeShort()`
- ✓ `char writeChar()`
- ✓ `int writeInt()`
- ✓ `float writeFloat()`
- ✓ `writeDouble()`
- ✓ `writeBytes()`
- ✓ `writeChars()`
- ✓ `writeUTF()`



java.io.DataOutputStream (3)



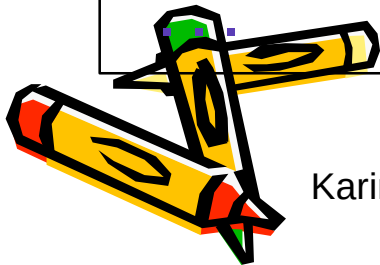
○ Fermeture de flot

- ✓ la fermeture se fait avec la méthode `close()`
- ✓ la fermeture de **DataOutputStream** entraine la fermeture de **InputStream** original

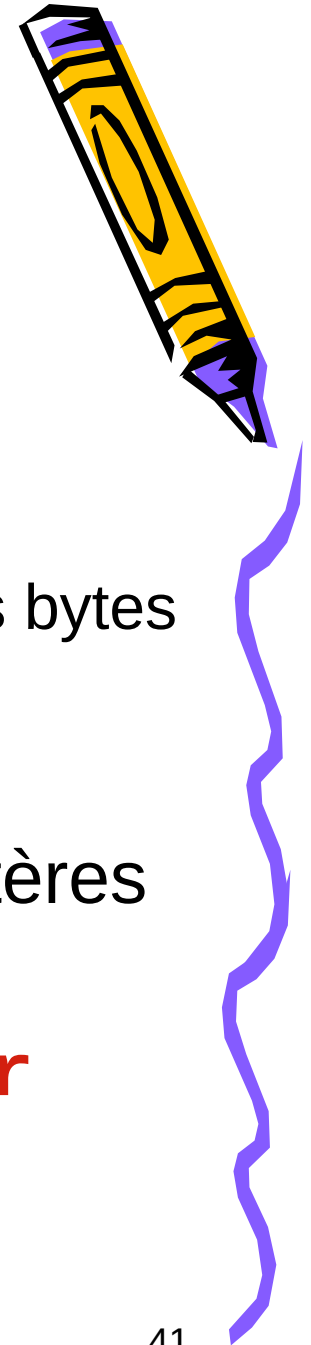


java.io.DataOutputStream (4)

```
FileOutputStream fos =  
    new FileOutputStream("toto.txt");  
BufferedOutputStream bos =  
    new BufferedOutputStream(fos);  
DataOutputStream dos = new DataOutputStream(bos);  
dos.writeChars("Position ");  
dos.writeDouble(10.0);  
dos.writeDouble(12.0);  
dos.writeDouble(7.0);  
dos.writeChars("\n");
```



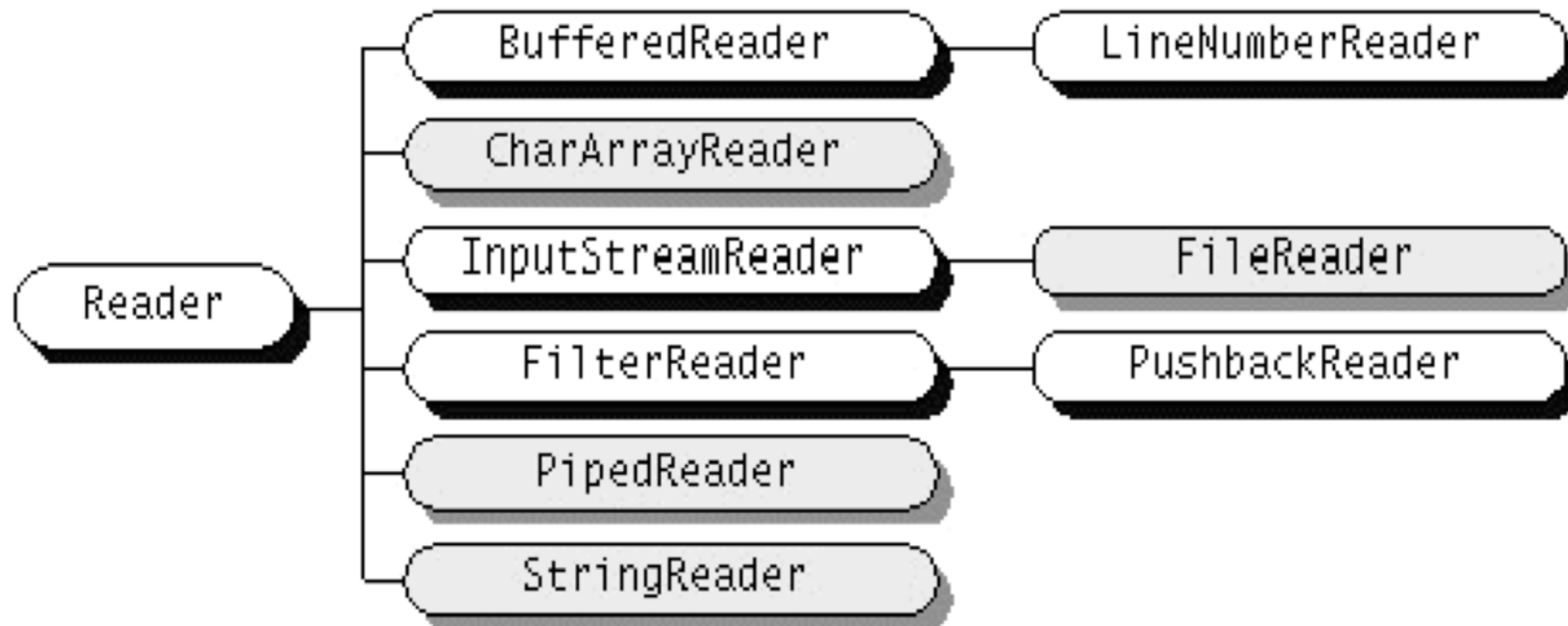
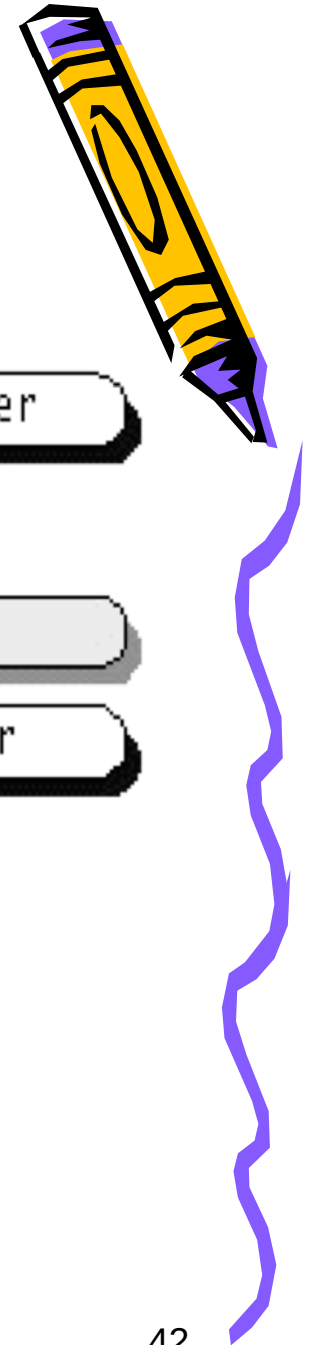
Flots de caractères (1)



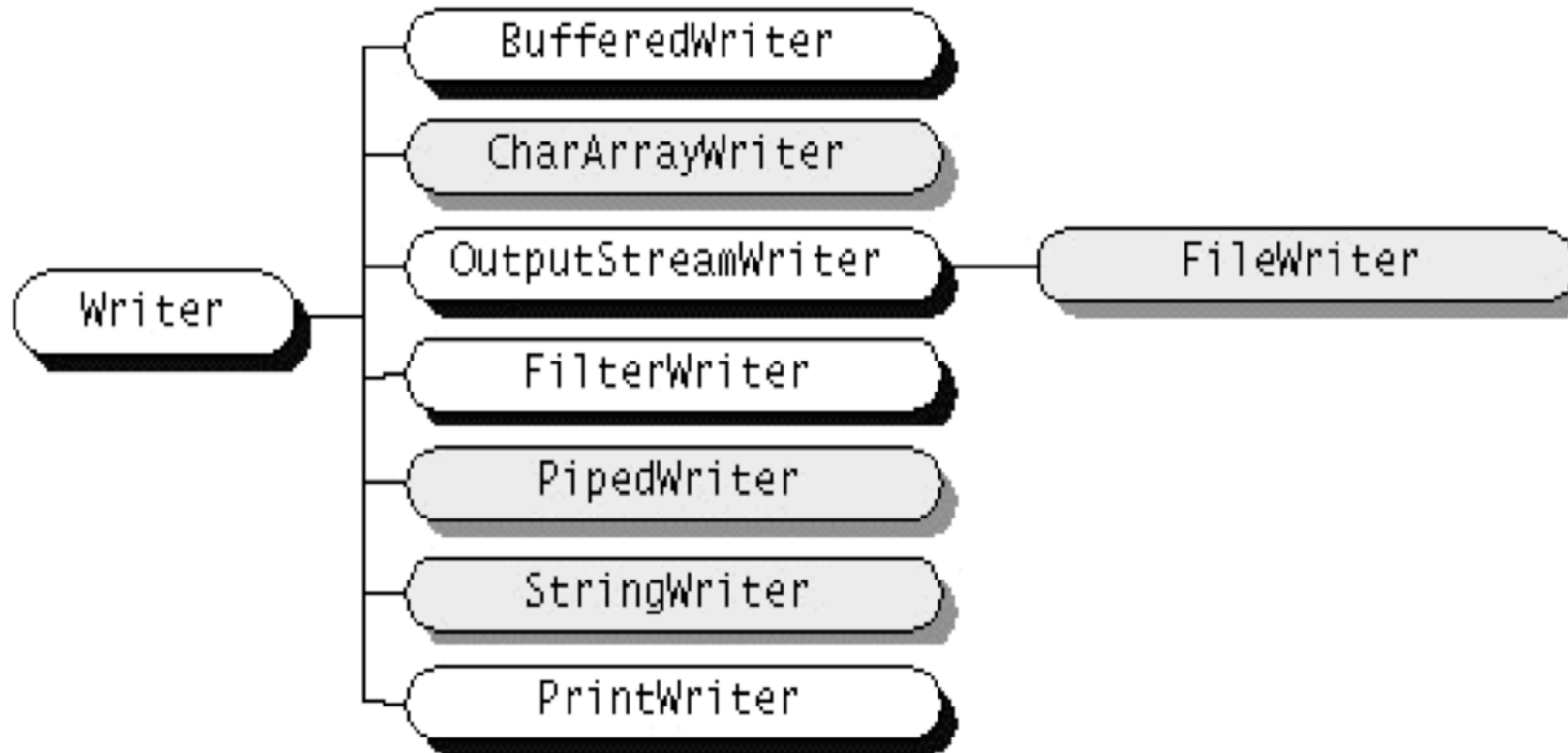
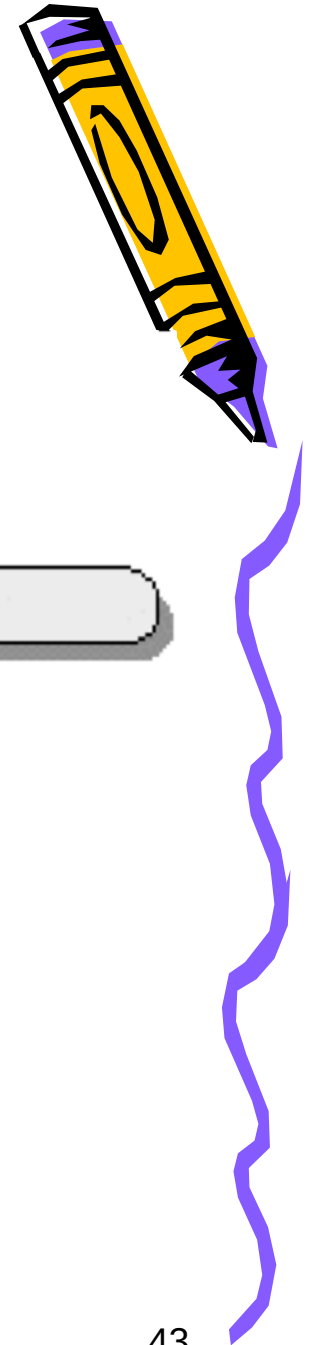
- les flots de caractères (*character streams*)
 - ✓ ils existent depuis la JDK1.1
 - ✓ auparavant
 - ▲ les entrées/sorties ne travaillaient qu'avec des bytes
 - ▲ Il n'y avait que les classes **InputStream** et **OutputStream**
- les flots de caractères manipulent des caractères au format unicode (16 bits)
- il y a 2 classes principales **Reader** et **Writer**



Flots de caractères (2)

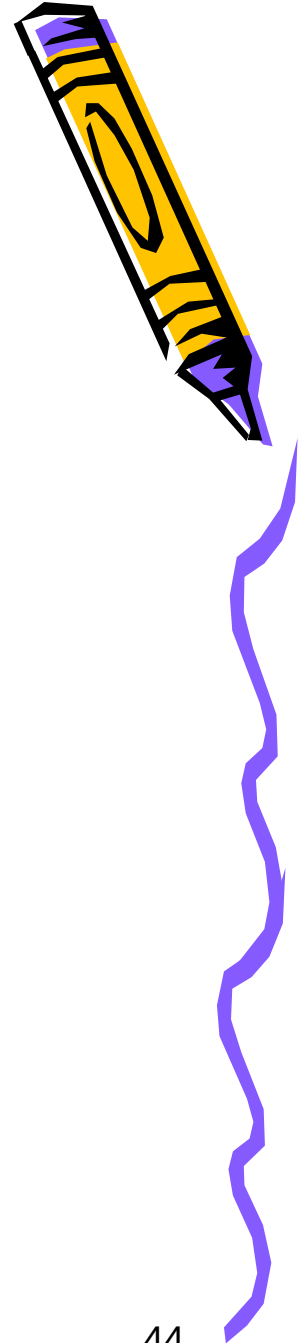


Flots de caractères (3)

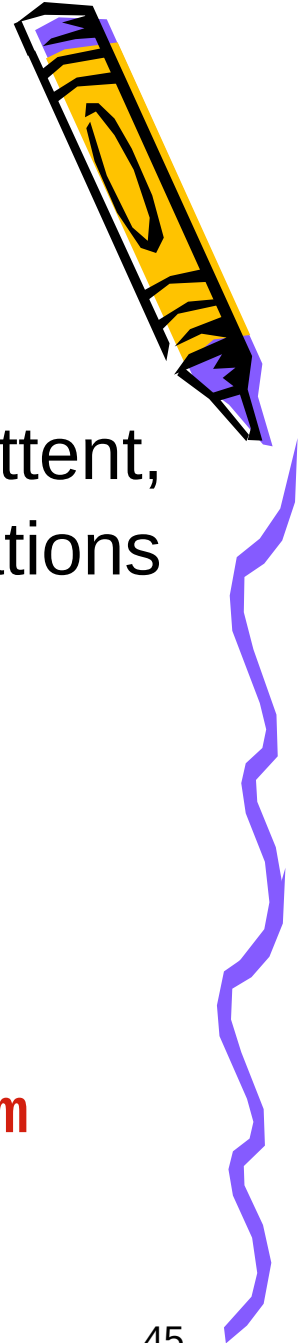


Flots de caractères (4)

- les classes les plus utilisées sont
 - ✓ **InputStreamReader**
 - ✓ **BufferedReader**
 - ✓ **FileReader**
 - ✓ **PrintWriter**
 - ✓ **FileWriter**



Flots de caractères (5)



○ Classes Reader et Writer

- les classes **Reader** et **Writer** permettent, globalement, de faire les mêmes opérations que les classes **InputStream** et **OutputStream**
- elles supportent des fonctionnalités similaires
 - ✓ **FileReader** <--> **FileInputStream**
 - ✓ **FileWriter** <--> **FileOutputStream**



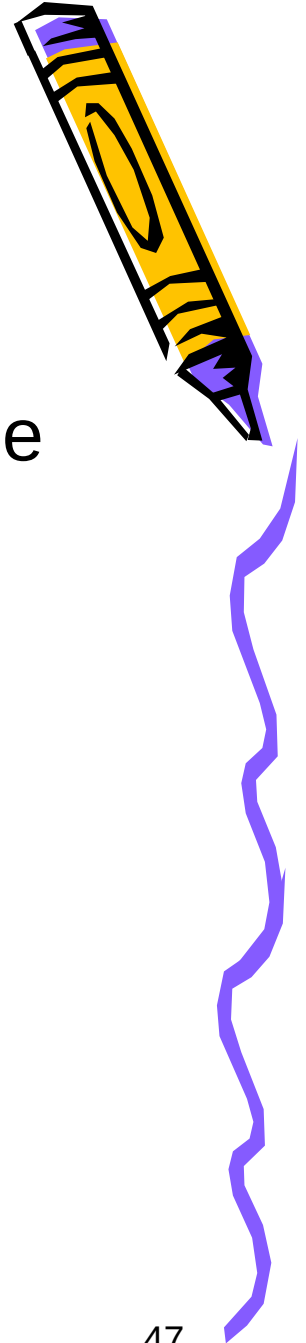
Flots de caractères (6)



- en utilisant les **flots de caractères**, on peut échanger des données textuelles...
 - ✓ avec des accents
 - ✓ caractères bizarres (japonais...)
 - ✓ ...
- **InputStreamReader** et **OutputStreamWriter** se chargent de l'encodage et du décodage
 - ✓ ce sont toujours des octets qui circulent



Flots de caractères (7)



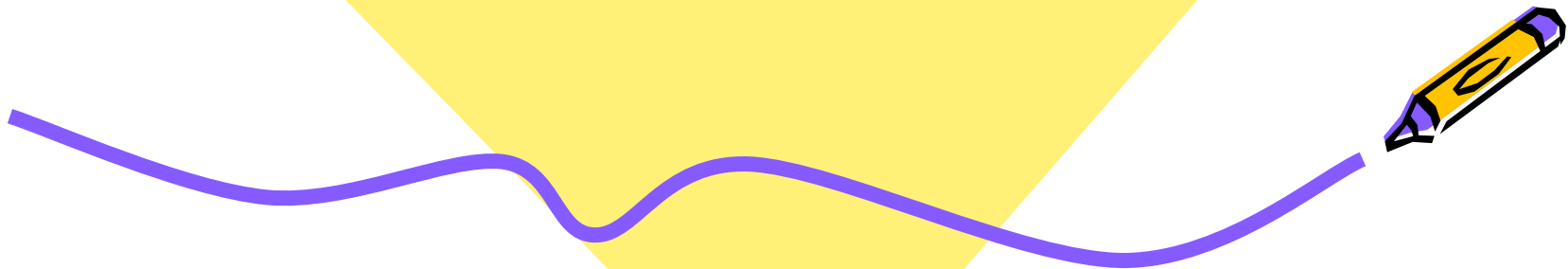
- les flots de caractères sont plus efficaces que **InputStream** et **OutputStream**
- les classes **InputStreamReader** et **OutputStreamWriter** sont bufférisés en standard



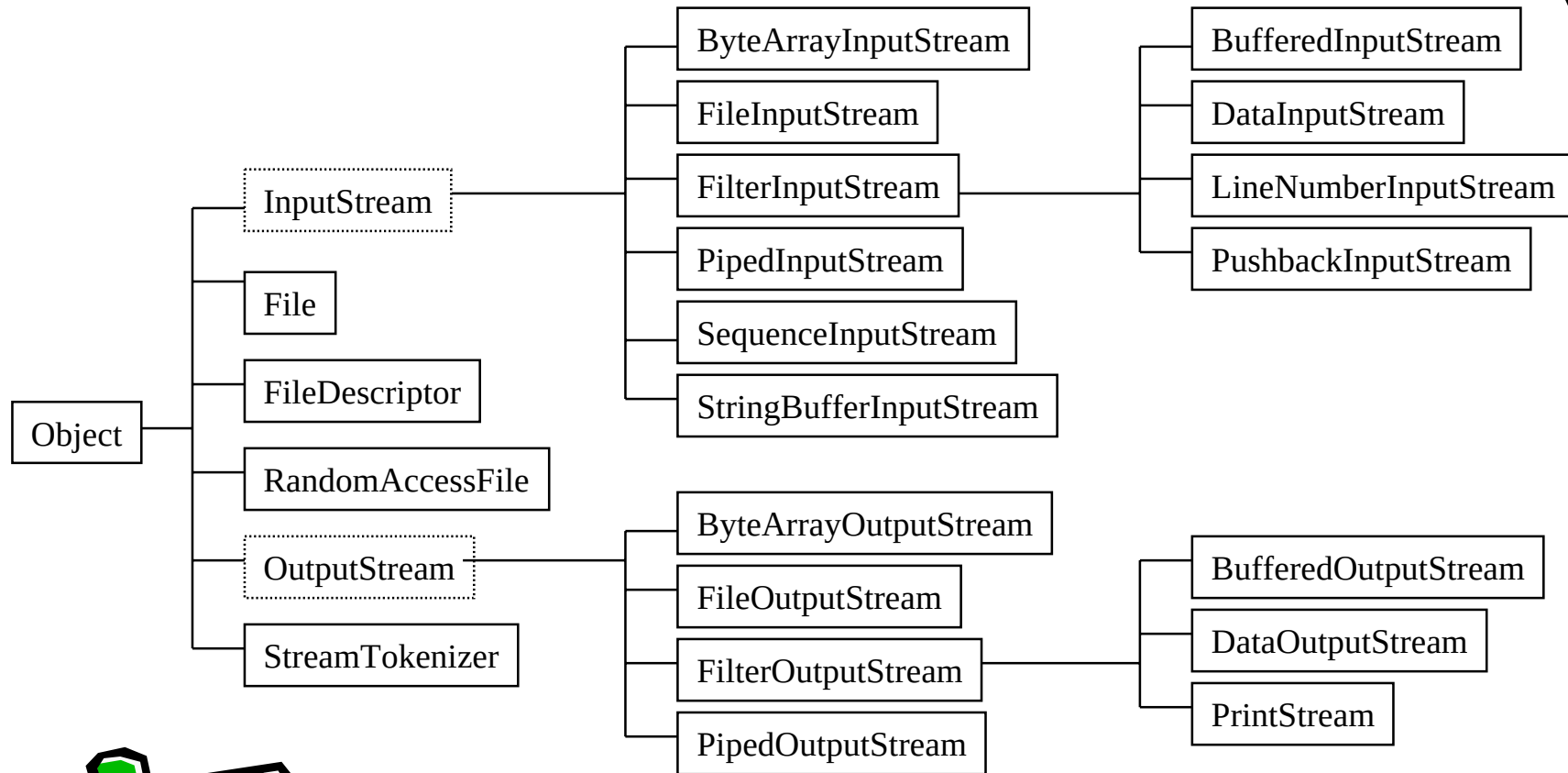
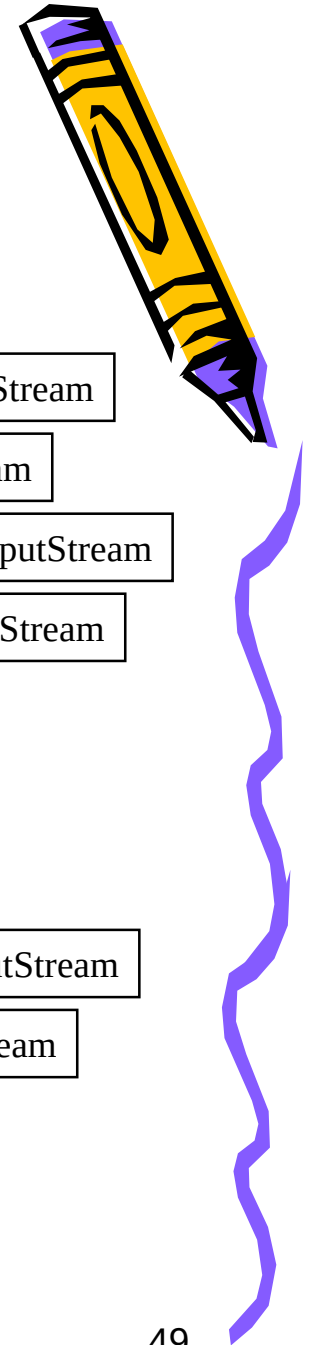


Complément sur les E/S

Karima Boudaoud



Package java.io.*



java.io.File

- Cette classe fournit une définition *platform-independent* des fichiers et des répertoires

```
File f = new File("/etc/passwd");  
System.out.println(f.exists());    // --> true  
System.out.println(f.canRead());   // --> true  
System.out.println(f.canWrite());  // --> false  
System.out.println(f.getLength()); // --> 11345
```

```
File d = new File("/etc/");  
System.out.println(d.isDirectory()); // --> true
```

```
String[] files = d.list();  
for(int i=0; i < files.length; i++)  
    System.out.println(files[i]);
```



java.io.File(Input|Output) Stream

- Ces classes permettent d'accéder en lecture et en écriture à un fichier

```
FileInputStream fis = new FileInputStream("source.txt");  
byte[] data = new byte[fis.available()];  
fis.read(data);  
fis.close();
```

```
FileOutputStream fos = new FileOutputStream("cible.txt");  
fos.write(data);  
fos.close();
```



java.io.Data(Input | Output) Stream

- Ces classes permettent de lire et d'écrire des types primitifs et des lignes sur des flux

```
FileInputStream fis = new FileInputStream("source.txt");  
DataInputStream dis = new DataInputStream(fis);
```

```
int i    = dis.readInt();  
double d = dis.readDouble();  
String s = dis.readLine();
```

```
FileOutputStream fos = new FileOutputStream("cible.txt");  
DataOutputStream dos = new DataOutputStream(fos);
```

```
dos.writeInt(123);  
dos.writeDouble(123.456);  
dos.writeChars("Une chaine");
```

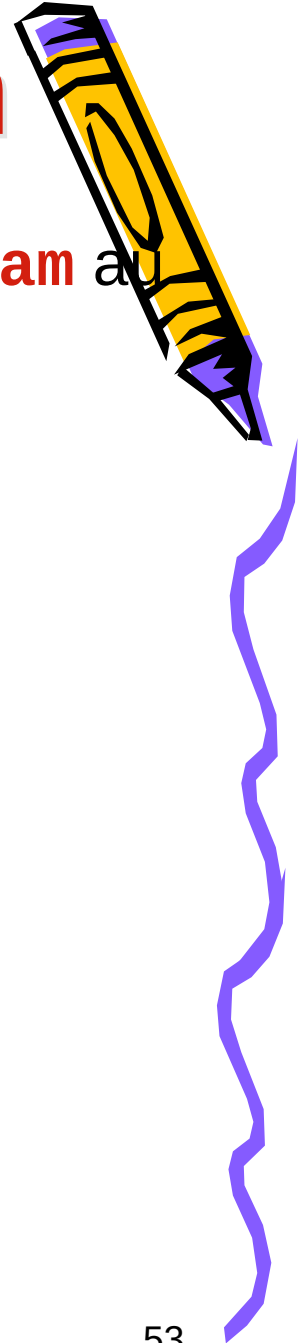


java.io.PrintStream

- Cette classe permet de manipuler un **OutputStream** au travers des méthode **print()** et **println()**

```
PrintStream ps = new PrintStream(new  
    FileOutputStream("cible.txt"));
```

```
ps.println("Une ligne");  
ps.println(123);  
ps.print("Une autre ");  
ps.print("ligne");  
ps.flush();  
ps.close();
```



java.io.Object(Input|Output) Stream (1)

- Ces classes permettent de lire et d'ecrire des objets, implémentant **java.io.Serializable**, sur des flux.

// Ecriture

```
FileOutputStream fos = new FileOutputStream("tmp");  
ObjectOutput oos = new ObjectOutputStream(fos);  
oos.writeObject("Today");  
oos.writeObject(new Date());  
oos.flush();
```

// Lecture

```
FileInputStream fis = new FileInputStream("tmp");  
ObjectInputStream ois = new ObjectInputStream(fis);  
String today = (String)ois.readObject();  
Date date = (Date)ois.readObject();
```



java.io.Object (Input | Output) Stream (2)

- par défaut, tous les champs sont sérialisés (y compris **private**)
- cela peut poser des problèmes de sécurité
- il existe 3 solutions
 - ✓ ne pas implémenter **Serializable**
 - ✓ réécrire les méthodes **writeObject()** et **readObject()**
 - ✓ le mot clé **transient** permet d'indiquer qu'un champs ne doit pas être serialisé

