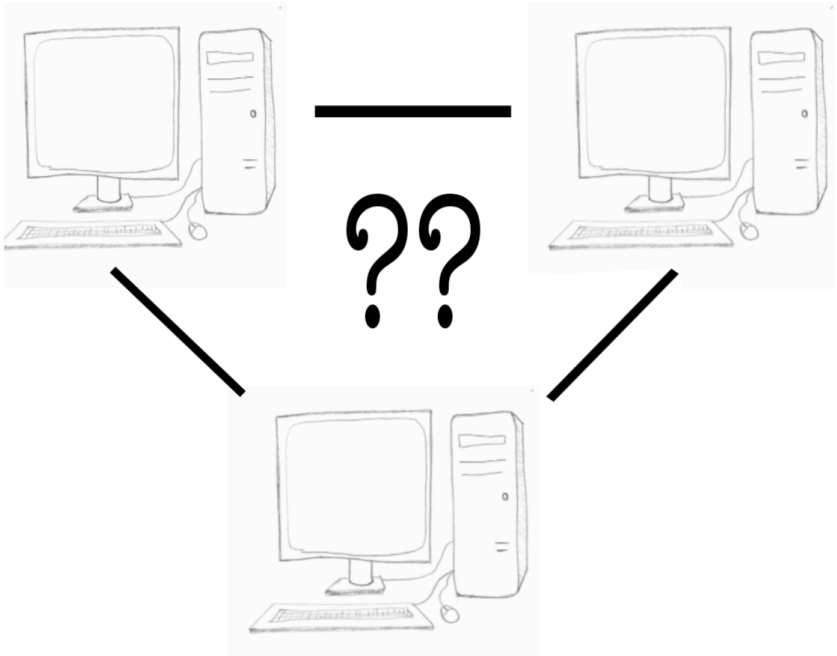
PROGRAMMATION RÉSEAU

Arnaud Sangnier
sangnier@irif.fr

API TCP Java - I



Différence flux et paquet

- Dans la communication **par flux** (comme **TCP**)
  - Les informations sont reçues dans l'ordre de leur émission
  - Il n'y a pas de perte
  - Inconvénient :
    - Établissement d'une connexion
    - Nécessité de ressources supplémentaire pour la gestion
- Dans la communication **par paquet** (comme **UDP**)
  - Pas d'ordre dans la délivrance des paquets
    - Un paquet posté en premier peut arrivé en dernier
  - Pas de fiabilité
    - Un paquet envoyé peut être perdu

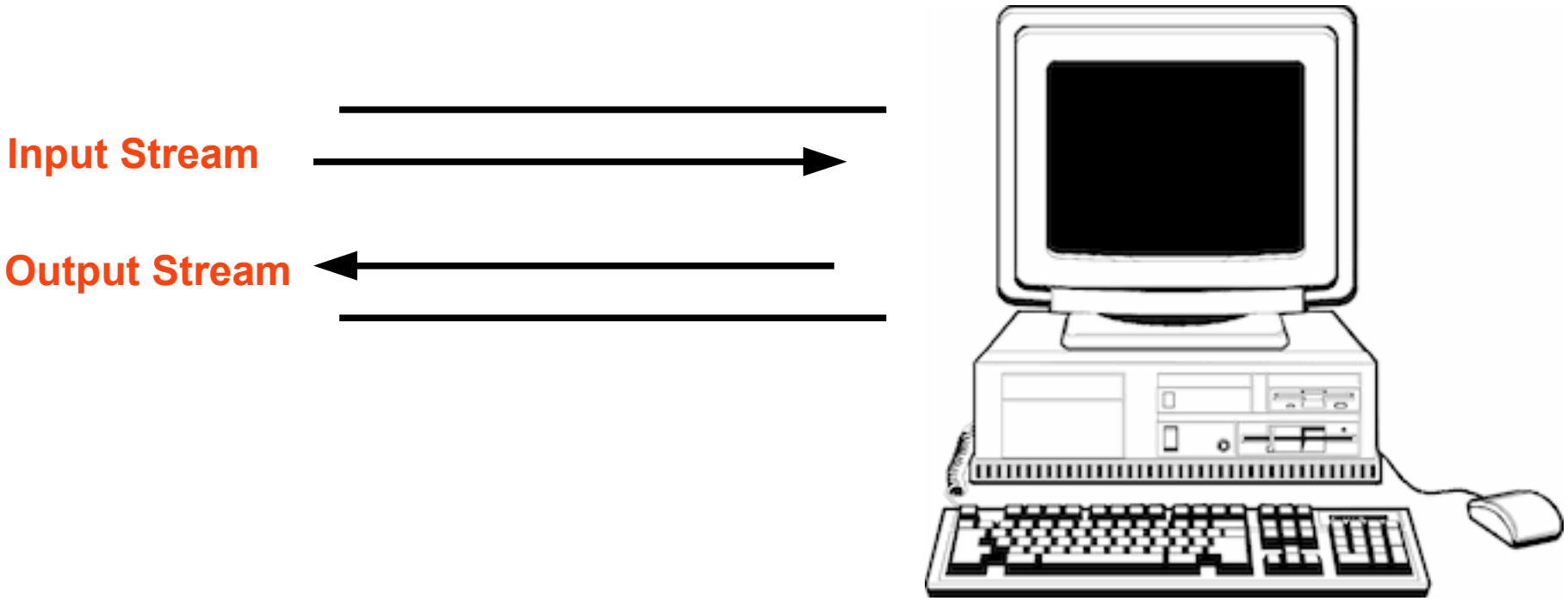
Aujourd'hui

- Comment communiquer en **Java** en TCP
  - Comment se connecter à un service (présent sur une **machine** et écoutant sur un **port**)
  - Comment lire les messages envoyés par ce service
  - Comment envoyer des messages à ce service
  - En d'autres termes, comment créer un **client TCP**
- Comment créer un service qui écoute sur un port donné de la machine où l'on se trouve
  - En d'autres termes, comment créer un **serveur TCP**

Entrées/Sorties en Java

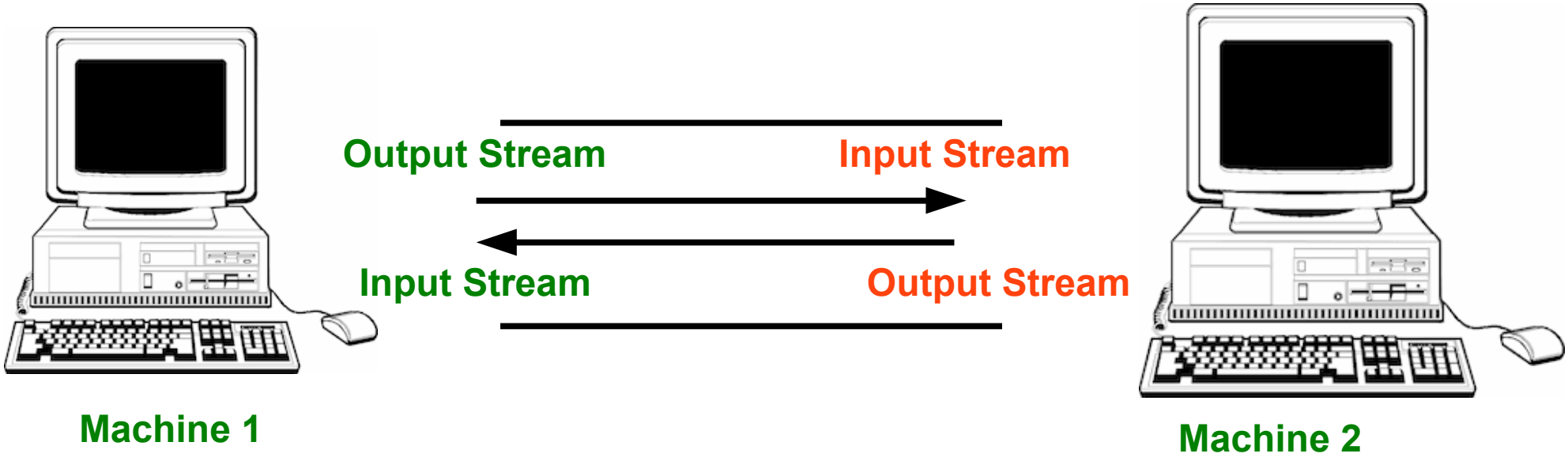
- Les entrées/sorties en Java sont construits sur des **flux** (***streams***)
  - les ***Input stream*** permettent de lire des données
  - les ***Output stream*** permettent d'écrire des données
- Il existe différentes classes de flux selon certaines classes de données
  - par exemple :
    - **java.io.FileInputStream**
    - **sun.net.TelnetOutputStream**
- Tous les Input stream utilisent des méthodes similaires de lecture
- Tous les Output stream utilisent des méthodes similaires d'écriture
- On utilise des filtres de flux pour manipuler plus facilement les données
  - par exemple pour manipuler des **String** plutôt que des **byte[]**

Les flux en réseau



Sur une machine, le flux d'Input sont les données qui arrivent et le flux Output, les données qui partent

Les flux en réseau



Les données qui sortent sur le flux de sortie de la Machine 1 arrivent sur le flux d'entrée de la Machine 2 !

Synchronisation

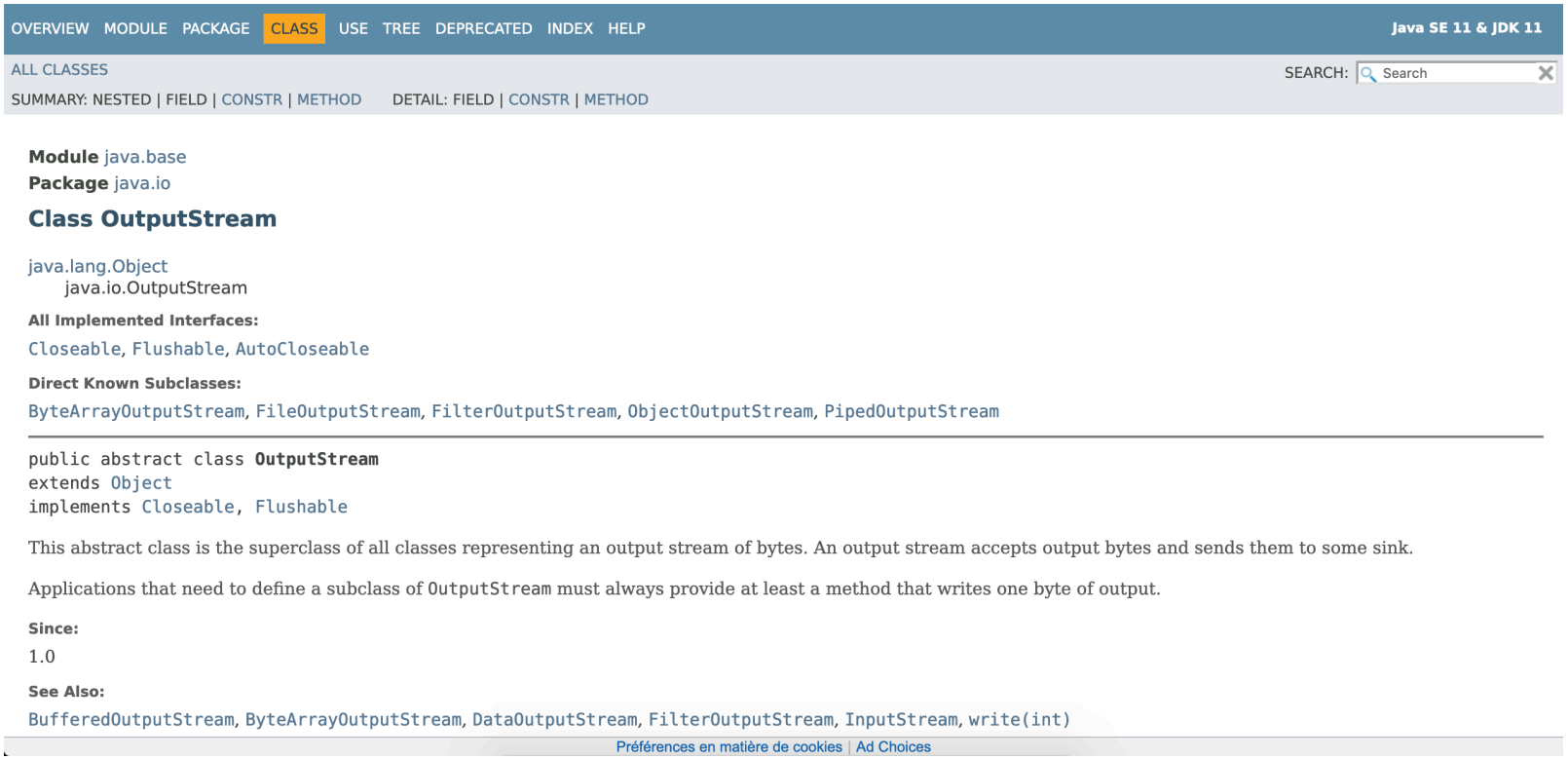
- Les flux sont **synchronisés**
  - Quand un programme demande de lire une donnée sur un flux, il **bloque** jusqu'à ce qu'une donnée soit disponible sur le flux
- Il est possible de faire en java des entrées/sorties non-bloquantes
  - C'est plus compliqué à utiliser
  - On s'en sert souvent pour des questions d'efficacité
  - Pour la plupart des applications que vous développerez, ce n'est pas nécessaire

Flux de sortie (*Output Stream*)

- La classe de base pour les flux de sortie :
  - **java.io.OutputStream** (classe abstraite)
- Elle utilise les méthodes suivantes :
  - **public abstract void write(int b) throws IOException**
    - prend en entrée un entier b entre 0 et 255 et écrit sur le flux l'octet correspondant
  - **public void write(byte[] data) throws IOException**
  - **public void write(byte[] data, int offset, int length) throws IOException**
  - **public void flush() throws IOException**
  - **public void close() throws IOException**

Utilisez la documentation

- Pour mieux comprendre, n'hésitez pas à consulter la documentation Java
- Sur les machines de l'Ufr, Java 1.11 est installé
- **https://docs.oracle.com/en/java/javase/11/docs/api/**



OVERVIEW MODULE PACKAGE CLASS USE TREE DEPRECATED INDEX HELP — Java SE 11 & JDK 11

ALL CLASSES — SEARCH: Search

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Module java.base
Package java.io

Class OutputStream

java.lang.Object
    java.io.OutputStream

All Implemented Interfaces:

Closeable, Flushable, AutoCloseable

Direct Known Subclasses:

ByteArrayOutputStream, FileOutputStream, FilterOutputStream, ObjectOutputStream, PipedOutputStream

---

```
public abstract class OutputStream
extends Object
implements Closeable, Flushable
```

This abstract class is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

Applications that need to define a subclass of `OutputStream` must always provide at least a method that writes one byte of output.

Since:

1.0

See Also:

BufferedOutputStream, ByteArrayOutputStream, DataOutputStream, FilterOutputStream, InputStream, write(int)

Préférences en matière de cookies | Ad Choices

Flush et close

- Les flux peuvent être bufferisés
  - Cela signifie que les données passent par des buffers
  - Ainsi quand on a fini d'écrire des données il est important de faire un **flush** sur le flux
  - Par exemple :
    - Si on envoie 300 octets à serveur et on attend sa réponse avant de continuer, et si le buffer fait 1024 octets, il se peut que le flux attende de remplir le buffer avant d'envoyer les données
    - L'action **flush** permet d'éviter ce souci
  - L'action **flush** permet de vider le buffer
  - Si on ferme un flux avec **close** sans faire de **flush**, des données peuvent être perdues
- L'action **close** ferme le flux et libère les ressources associées
  - comme les descripteurs de fichiers ou les ports

Flux d'entrée (*Input Stream*)

- La classe de base pour les flux d'entrée :
  - **java.io.InputStream** (classe abstraite)
- Elle utilise les méthodes suivantes :
  - **public abstract int read() throws IOException**
    - lit un entier de données et renvoie l'entier correspondant
  - **public int read(byte[] input) throws IOException**
  - **public int read(byte[] input, int offset, int length) throws IOException**
  - **public long skip(long n) throws IOException**
  - **public int available() throws IOException**
  - **public void close() throws IOException**

Un point sur les lectures

- La méthode **public int read(byte[] input) throws IOException**
  - elle lit des octets sur le flux et remplit le tableau **input**
  - elle ne remplit pas nécessairement **input** en entier !!!
- Par exemple , si ***in*** est un objet de la classe **InputStream** :

```
byte[] input=new byte[1024];
int bytesRead=in.read(input);
```

  - Si seulement 512 octets sont disponibles sur le flux, la méthode **read** remplira 512 cases du tableau **input** et renverra l'entier 512
  - Il se peut que 512 autres octets arrivent ensuite ou par morceau comment fait-on dans ce cas ?

Un point sur les lectures (2)

```
int bytesRead = 0;
int bytesToRead = 1024;
byte[] input=new byte[bytesToRead];
while(bytesRead < bytesToRead){
  bytesRead = bytesRead +
              in.read(input, bytesRead, bytesToRead - bytesRead);
}
```

- Ci-dessus on a mis le read dans une boucle pour remplir le tableau
- Cette boucle lit sur le flux in jusqu'à ce que le tableau soit rempli
- On utilise la méthode **public int read(byte[] input, int offset, int length) throws IOException**
  - Cette méthode remplit **length** octet du tableau **input** à partir de la position **offset**
  - Que se passe-t-il si 1024 octets n'arrivent jamais dans le flux ?
  - **Le code ci-dessus a un problème**

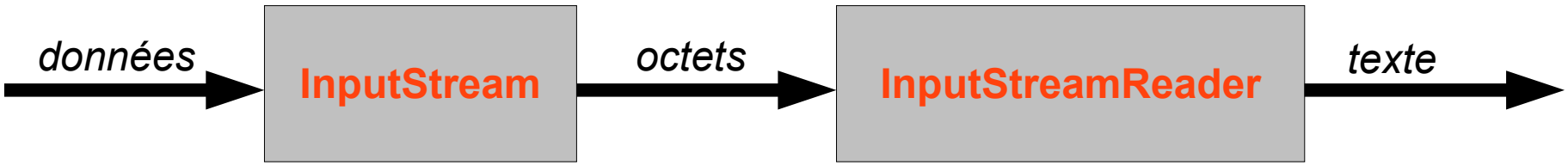
Un point sur les lectures (3)

```
int bytesRead = 0;
int bytesToRead = 1024;
byte[] input=new byte[bytesToRead];
while(bytesRead < bytesToRead){
  result=in.read(input,bytesRead,bytesToRead - bytesRead);
  if(result == -1) break;
  bytesRead = bytesRead + result;
}
```

- On résout le problème précédent en testant si la lecture renvoie -1
- Dans ce cas, cela signifie que le flux est terminé
- Quand le read renvoie -1, rien n'est mis dans le tableau **input**

Des octets aux caractères

- Les classes **InputStream** et **OutputStream** manipulent des octets
- On peut utiliser des filtres pour manipuler directement des caractères à la place des octets
- On ainsi les deux classes **InputStreamReader** et **OutputStreamWriter** qui permettent de prendre des caractères en entrée et les "passe" ensuite aux **InputStream** et **OutputStream** correspondant
- Cela permet de manipuler des caractères plutôt que des octets

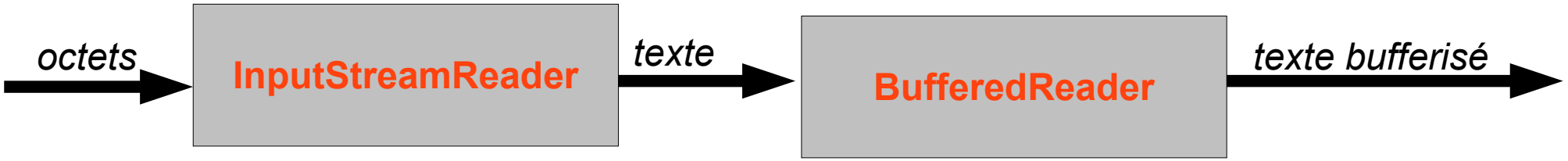


Lecteur

- La lecture en caractère se fait par un lecteur (Reader)
- Exemple de lecteur utilisé :
  - la classe **java.io.InputStreamReader**
  - cette classe étend la classe abstraite **Reader**
- On passe au constructeur un objet de la classe **InputStream**
  - **InputStreamReader(InputStream in)**
- Les méthodes que l'on peut utiliser :
  - **public void close()**
  - **public int read() throws IOException**
    - lit un seul caractère et retourne son code unicode (entre 0 et 65535)
  - **public int read(char[] cbuf,int offset,int length) throws IOException**
    - remplit le tableau **cbuf** en commençant à la position **offse**t avec **lengt**h caractères
    - renvoie le nombre de caractères lus

Lecteur amélioré

- La lecture caractère par caractère peut être fastidieuse
- On peut bufferisé la lecture en rajoutant une classe filtre



- On utilise la classe **BufferedReader**
  - **BufferedReader(Reader in)**
- Avec les mêmes méthodes que dans InputStreamReader
  - **public void close()**
  - **public int read() throws IOException**
  - **public int read(char[] cbuf,int offset,int length) throws IOException**
- et en plus :
  - **public String readLine()**

Résumé sur les lectures

- **InputStream** : pour lire des octets
- **InputStreamReader** : pour lire des caractères
- **BufferedReader** : pour lire des caractères bufferisés (permet de lire des lignes)
- Par exemple : si **in** est un objet de la classe **InputStream**

```
BufferedReader bf=new BufferedReader(
                          new InputStreamReader(in));
String s=bf.readLine()
```

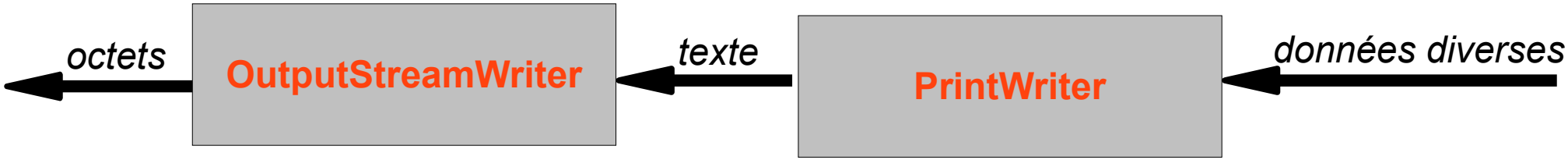
- Les méthodes de lecture sont là aussi bloquantes
- **close()** sur le **BufferedReader** ferme le flux correspondant

Écrivain

- L'écriture en caractère se fait par un écrivain (Writer)
- Exemple de lecteur utilisé :
  - la classe **java.io.OutputStreamWriter**
  - cette classe étend la classe abstraite **Writer**
- On passe au constructeur un objet de la classe **OutputStream**
  - **OutputStreamWriter(OutputStream out)**
- Les méthodes que l'on peut utiliser :
  - **public void close() throws IOException**
  - **public void flush() throws IOException**
  - **public void write(int c) throws IOException**
    - écrit un seul caractère donné par son code unicode
  - **public void write(char[] cbuf, int off, int len) throws IOException**
    - écrit **len** caractères de **cbuf** en commençant par **off**
  - **public void write(String str, int off, int len) throws IOException**

Écrivain amélioré

- L'écriture caractère par caractère peut être fastidieuse
- On peut utilisé au dessus un objet **PrintWriter** pour rendre les choses plus agréables



- **PrintWriter(Writer out)**
- Méthodes sur le flux sous-jacent
  - **public void close() throws IOException**
  - **public void flush() throws IOException**
- Plein de méthodes utiles pour l'écriture (convertissant les données en texte) :
  - **public void print(char c), public void print(int i), public void print(String s), public void println(String s), public void print(Object o),....**
- N 'hésitez pas à regarder la documentation

Résumé sur les écritures

- **OutputStream** : pour écrire des octets
- **OutputStreamWriter** : pour écrire des caractères
- **PrintWriter** : pour écrire tout type de données
- Par exemple : si **out** est un objet de la classe **OutputStream**

```
PrintWriter pw=new PrintWriter(
                        new OutputStreamWriter(out));
pw.println("Hello");
```

- Là aussi il ne faut pas oublier de faire **flush()**
- **close()** sur le **PrintWriter** ferme le flux correspondant