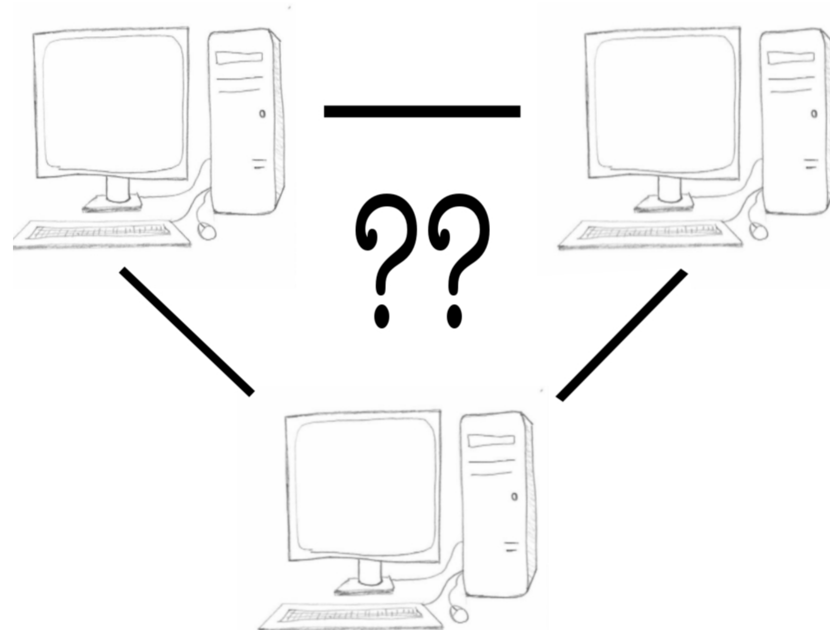
PROGRAMMATION RÉSEAU

Arnaud Sangnier
sangnier@irif.fr

API TCP Java



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

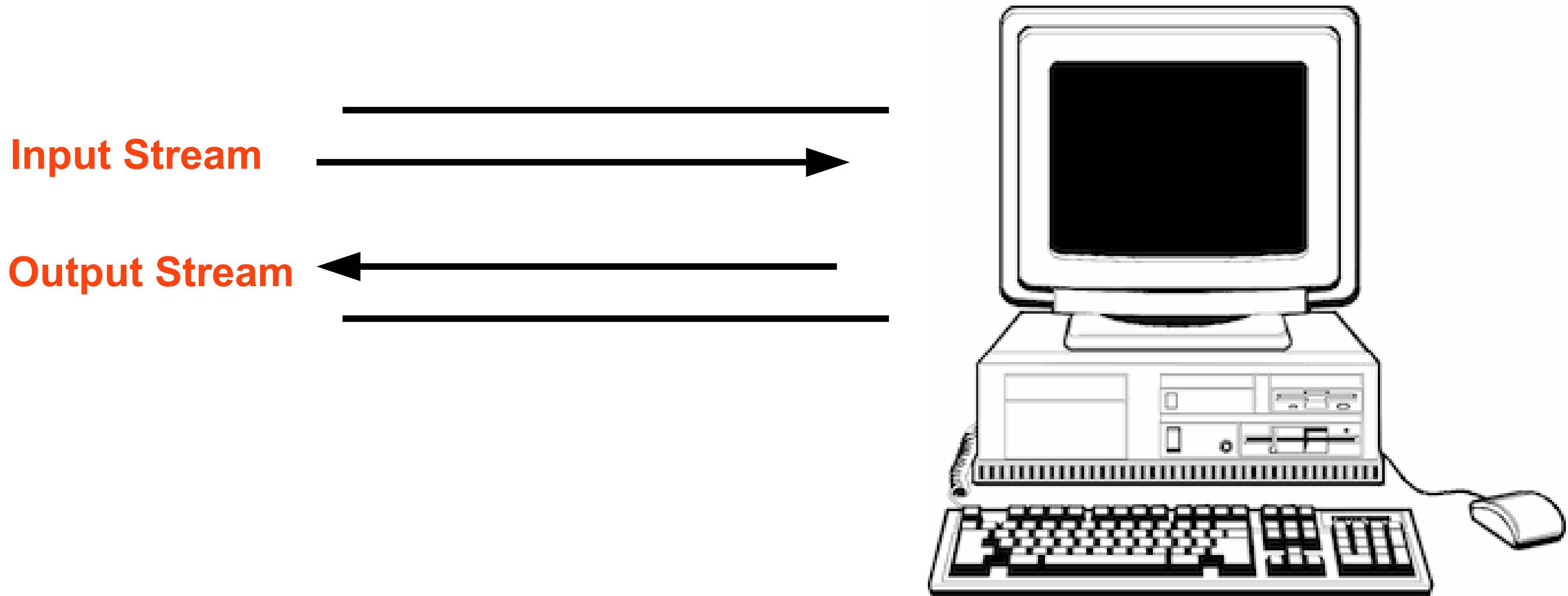
Aujourd'hui

- 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

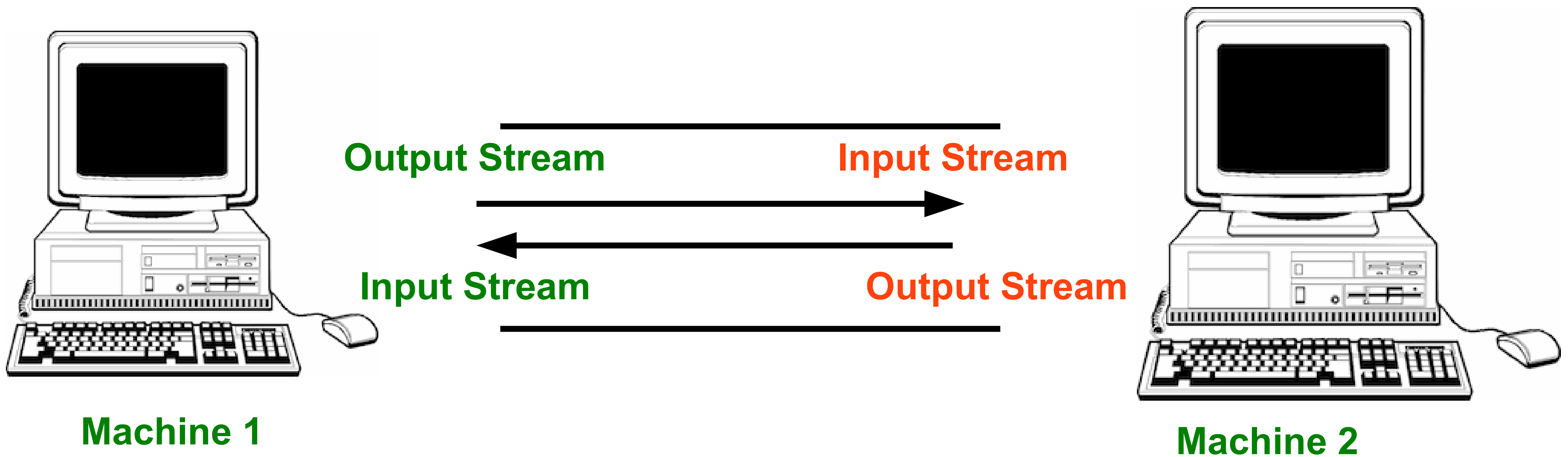
- Les entrées/sorties en Java sont construits sur des **flux** (***streams***)
  - les ***Input stream*** permettent de lire des données
  - les ***Output stream*** permettent d'écrire des données
- Il existe différentes classes de flux selon certaines classes de données
  - par exemple :
    - **java.io.FileInputStream**
    - **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

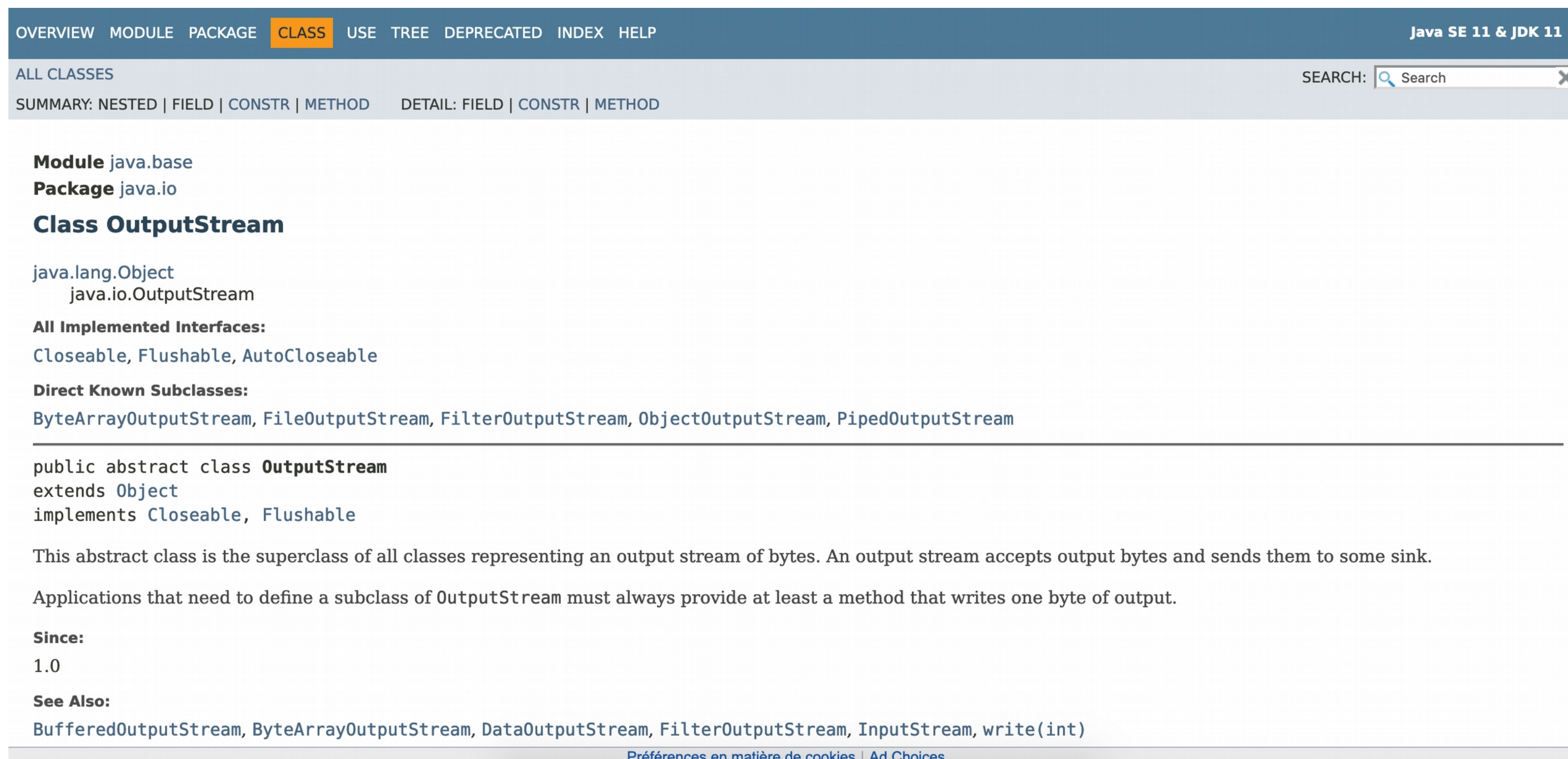
- Les flux sont **synchronisés**
  - Quand un programme demande de lire une donnée sur un flux, il **bloque** jusqu'à ce qu'une donnée soit disponible sur le flux
- Il est possible de faire en java des entrées/sorties non-bloquantes
  - C'est plus compliqué à utiliser
  - On s'en sert souvent pour des 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/



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**

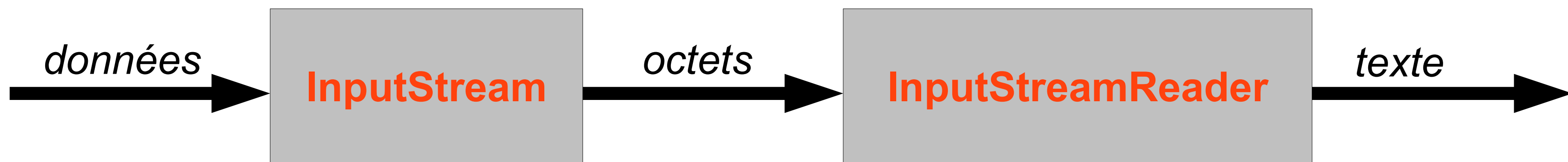
Un point sur les lectures (3)

```
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

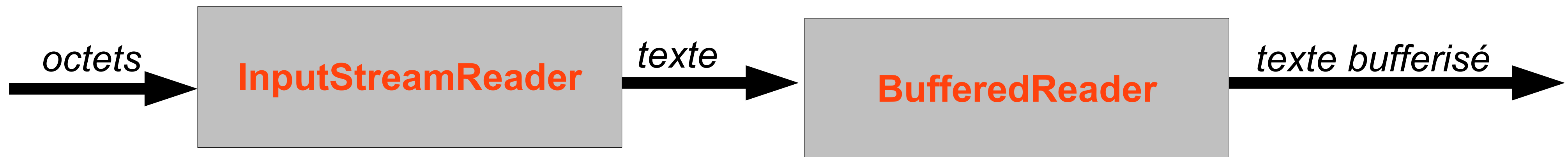


Lecteur

- 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

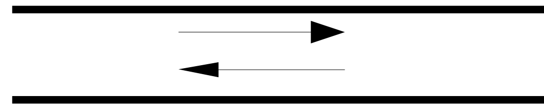
- **OutputStream** : pour écrire des octets
- **OutputStreamWriter** : pour écrire des caractères
- **PrintWriter** : pour écrire tout type de données
- Par exemple : si **out** est un objet de la classe **OutputStream**

```
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

Retour sur notre problème

- **TCP -> mode connecté**



Client **Serveur**

- Comment se connecter à un service ?
- Comment envoyer nos données vers un service ?
- Comment recevoir les données envoyées ?

Liens avec les flux et le réseau

Pourquoi on vient de
passer tout ce temps
à parler de flux ?

- Il existe un outil, les **sockets** qui permet de se connecter à des machines et de faire passer toutes les communications avec des flux
- Donc c'est sur des flux que vous enverrez et recevrez des données
- Les fonctions vues précédemment vont donc être très utiles

Les sockets en bref

- Une socket est une connexion entre deux machines
- Elle passe donc par deux ports
- Opérations basiques pouvant être réalisées par une socket
  - Se connecter à une machine à distance
  - Envoyer des données
  - Recevoir des données
  - Fermer une connexion
  - Se connecter à un port
  - Attendre des données
  - Accepter des connexions de machines sur le port auquel elle est liée
- Les trois dernières opérations sont nécessaires pour les serveurs (cf la classe **ServerSocket**)

Client pour le service echo tcp

- Questions à se poser avant de programmer le client :
  - Sur quelle machine tourne le service avec lequel on souhaite communiquer
    - **lampe.informatique.univ-paris-diderot.fr**
  - Sur quel port écoute ce service
    - **port 7**
    - on peut retrouver cette information dans **/etc/services**
  - Quelle forme a la communication
    - On envoie des chaînes de caractères
    - Le client commence à 'parler'
    - Le serveur renvoie la même chaîne

Création de socket TCP

- On va créer une socket pour se connecter au port 7 de monjetas
- Utilisation de la classe **java.net.Socket**
  - Plusieurs constructeurs possibles (à explorer)
  - On va utiliser le suivant :
    - **public Socket(String host, int port) throws UnknownHostException, IOException**
    - **host** est le nom de la machine distante
    - **port** est le numéro du port

```
Socket socket=new Socket("lampe.informatique.univ-paris-diderot.fr",7);
```

Création de socket TCP(2)

- Que se passe-t-il quand on fait :

```
Socket socket=new Socket("lampe.informatique.univ-paris-diderot.fr",7);
```

- Une socket est créé entre la machine locale et **lampe**
- La socket est attachée localement à un port éphémère (choisi par java)
- À la création de la socket, une demande de connexion est lancée vers **lampe** sur le **port 7**
- Si la connexion est acceptée, le constructeur termine normalement
- On peut ensuite utiliser la socket pour communiquer

Traiter correctement les exceptions

- La signature de la création de socket est :
  - **public Socket(String host, int port) throws UnknownHostException, IOException**
- En java, il est important de *catcher* les exceptions afin d'éviter qu'un programme crashe brutalement
- En programmation réseaux, c'est d'autant plus important car les problèmes peuvent être **multiples** :
  - mauvais numéro de port, hôte inconnu, problème de lecture ou d'écriture sur un flux

```
try{...}
catch(Exception e){
    System.out.println(e);
    e.printStackTrace();
}
```

Exemple fonctionnant sur lulu

```
import java.net.*;
import java.io.*;
public class ClientEcho{
   public static void main(String[] args){
      try{
        Socket socket=new Socket("lampe",7);
        socket.close();
      }
      catch(Exception e){
        System.out.println(e);
        e.printStackTrace();
      }
   }
}
```

Programme levant une exception sur lulu

```
import java.net.*;
import java.io.*;
public class ClientPb{
    public static void main(String[] args){
        try{
            Socket socket=new Socket("lulu",10);
            socket.close();
        }
        catch(Exception e){
            System.out.println(e);
            e.printStackTrace();
        }
    }
}
```

Question

Mais si j'ai pas le nom de
la machine
mais juste l'adresse IP

- Il existe différents constructeurs dans la classe **Socket**
- Encore, plus simple, l'adresse de lampe est `192.168.70.237`
-

```
Socket socket=new Socket("192.168.70.237",7);
```

Utilisation de la socket créée

- Pour mettre fin à la connexion (c-à-d pour raccrocher) :
  - Utilisation de la méthode **void close()**
- Comment communique-t-on par la socket
  - la méthode **public InputStream getInputStream() throws IOException**
    - elle permet de récupérer un flux d'entrée sur lequel on va lire
  - la méthode **public OutputStream getOutputStream() throws IOException**
    - elle permet de récupérer un flux de sortie sur lequel on va écrire
- **Remarques :**
  - Les communications sur la socket sont **bi-directionnelles**
  - On utilise la même socket pour recevoir et envoyer des messages
  - **Par contre deux flux différents**

Récupération des flux

- Partie de code montrant comment on récupère les informations

```
Socket socket=new Socket("lampe",7);
InputStream is=socket.getInputStream();
OutputStream os=socket.getOutputStream();
```

- On a vu que l'on pouvait installer des filtres sur les objets des classes InputStream et OutputStream pour manipuler plus facilement les entrées-sorties

```
Socket socket=new Socket("lampe",7);
BufferedReader br=new BufferedReader(
    new InputStreamReader(socket.getInputStream()));
PrintWriter pw=new PrintWriter(
   new OutputStreamWriter(socket.getOutputStream()));
```

Création d'un client TCP (1)

- On va créer un client pour le service echo tcp (port 7) tournant sur lampe
- Voilà les étapes que notre client va faire
  1) Créer une socket pour se connecter au service
  2) Récupérer les flux d'entrée et sortie et les filtrer
  3) Envoyer une chaîne de caractères `"Hello"`
  4) Attendre et recevoir une chaîne de caractères du service
  5) Afficher la chaîne de caractères reçue
  6) Fermer les flux et la connexion

Création d'un client TCP (2)

```
import java.net.*;
import java.io.*;
public class ClientEcho{
    public static void main(String[] args){
        try{
            Socket socket=new Socket("lampe",7);
            BufferedReader br=new BufferedReader(new InputStreamReader(socket.getInputStream()));
            PrintWriter pw=new PrintWriter(new OutputStreamWriter(socket.getOutputStream()));
            pw.print("HELLO\n");
            pw.flush();
            String mess=br.readLine();
            System.out.println("Message recu :"+mess);
            pw.close();
            br.close();
            socket.close();
        }
        catch(Exception e){
            System.out.println(e);
            e.printStackTrace();
        }
    }
}
```

Les points importants

- Ne pas oublier de *catcher* les exceptions
- Utiliser la méthode flush() pour vider le buffer et envoyer le message
- La lecture avec readLine() bloque tant que l'on ne reçoit pas de message
- Exemple de code ne fonctionnant pas (sans flush le message n'est pas envoyé)

```
Socket socket=new Socket("lampe",7);
BufferedReader br=new BufferedReader(new
                 InputStreamReader(socket.getInputStream()));
PrintWriter pw=new PrintWriter(new
                 OutputStreamWriter(socket.getOutputStream()));
pw.print("HELLO\n");
String mess=br.readLine();
System.out.println("Message recu :"+mess);
```

La suite ?

Et comment fait-on un service qui écoute sur un port ?

- C'est ce que nous allons voir maintenant avec la classe **ServerSocket**
- Ce qui est bien
  - à part l'attente d'une connexion sur un port
  - ensuite on aura des sockets que l'on manipulera comme pour les clients

Création de un serveur

- Pour créer un serveur on va se servir de la classe **java.net.ServerSocket**
- Là encore il existe différents constructeurs et méthodes à explorer
- Deux méthodes vont être intéressantes :
  - **public ServerSocket(int port) throws IOException**
    - Crée une socket qui écoute sur un port et ne sert qu'à attendre les demandes de connexions
    - **port** est le numéro du port

```
ServerSocket server=new ServerSocket(4242);
```

  - **public Socket accept() throws IOException**
    - Attend une connexion
    - Retourne une socket qui servira (comme pour les clients à la communication

Et le nom de la machine ?

Pourquoi on ne donne pas
de nom de machines
au constructeur ?

```
ServerSocket server=new ServerSocket(4242);
```

- En fait, c'est simple, le service se trouve sur la machine où vous exécutez le programme
- L'objet **server** n'a donc pas besoin de connaître le nom de la machine

Comportement d'un serveur

1) Création d'un objet **ServerSocket** qui est lié à un port
2) Attente de demande de connexion avec **accept**
3) Récupération de la Socket pour la communication
4) Récupération de flux d'entrée-sortie
5) Communication avec le client
6) Fermeture des flux et de la socket
7) Retour au point 2)

Exemple Serveur TCP

- On va programmer un serveur qui va
  - Attendre une connexion sur le port 4242
  - Envoyer un message `"Hi\n"`
  - Attendre un message du client
  - Afficher le message du client
  - Et recommencer à attendre une communication

Création d'un serveur TCP (2)

```
import java.net.*;
import java.io.*;
public class ServeurHi{
    public static void main(String[] args){
        try{
            ServerSocket server=new ServerSocket(4242);
            while(true){
                Socket socket=server.accept();
                BufferedReader br=new BufferedReader(new InputStreamReader(socket.getInputStream()));
                PrintWriter pw=new PrintWriter(new OutputStreamWriter(socket.getOutputStream()));
                pw.print("HI\n");
                pw.flush();
                String mess=br.readLine();
                System.out.println("Message recu :"+mess);
                br.close();
                pw.close();
                socket.close();
            } }
        catch(Exception e){
            System.out.println(e);
            e.printStackTrace();
        }
    }
}
```

Points importants

- Il y a donc différentes sockets dans notre serveur
  - Une **ServerSocket** qui sert uniquement à attendre les connexions
  - Une socket (de la classe **Socket**) par communication acceptée
- Là encore il ne faut pas oublier :
  - De *catcher* les exceptions et d'afficher les messages d'erreur correctement
  - De faire un **flush** pour les écritures afin de garantir que les messages sont bien envoyés
  - De fermer proprement les socket et les flux d'entrée-sortie
    - par exemple ne pas fermer la socket ServerSocket dans la boucle d'attente des connexions, vu que l'on en a besoin pour la prochaine connexion

Comment tester notre serveur

- Tout d'abord, rappelez vous que vous aurez au moins deux programmes
  1) Le programme du serveur
  2) Les programmes des clients
- DONC si vous voulez tester le tout il faudra **PLUSIEURS** terminaux !!!!
- Avant de programmer un programme, pourquoi ne pas faire un coup de **telnet** sur votre serveur
  - Pour le serveur que nous avons programmé, si il tourne sur **lulu**
    - **telnet lulu 4242**
    - Notre serveur enverra alors `"Hi\n"` que telnet affichera
    - Si nous tapons un message sur telnet suivi de Retour Chariot
    - Le serveur affichera notre message

Pour programmer notre client

- Cette fois-ci la machine est lulu
- Cette fois-ci le port est 4242

```
Socket socket=new Socket("lulu",4242);
```

- Qui commence par parler ?
  - Le Serveur, il envoie `"Hi\n"`
- Et ensuite ?
  - Le client doit envoyer un message se terminant par `"\n"`
  - En effet le serveur fait readLine(), il attend donc une chaîne de caractères se terminant par un retour à la ligne

Client TCP pour notre serveur

```
import java.net.*;
import java.io.*;
public class ClientHi{
    public static void main(String[] args){
        try{
            Socket socket=new Socket("lulu",4242);
            BufferedReader br=new BufferedReader(new InputStreamReader(socket.getInputStream()));
            PrintWriter pw=new PrintWriter(new OutputStreamWriter(socket.getOutputStream()));
            String mess=br.readLine();
            System.out.println("Message recu du serveur :"+mess);
            pw.print("HALLO\n");
            pw.flush();
            pw.close();
            br.close();
            socket.close();
        }
        catch(Exception e){
            System.out.println(e);
            e.printStackTrace();
        }
    }
}
```

Remarques

- Notre serveur n'accepte pas plusieurs connexions en même temps
  - Quand il a fait un **accept**, il communique avec un client et il doit avoir fini la communication pour parler avec un autre client
  - Nous verrons la prochaine fois comment modifier cela
- Un client qui commencerait par écrire au serveur et lire ensuite ce que le serveur fait fonctionnerait
  - MAIS mauvaise interprétation du protocole
  - On pourrait croire que le serveur répond "Hi" au "Hallo" du client

Mauvais client TCP pour notre serveur

```
import java.net.*;
import java.io.*;
public class ClientHiFalse{
    public static void main(String[] args){
        try{
            Socket socket=new Socket("lulu",4242);
            BufferedReader br=new BufferedReader(new InputStreamReader(socket.getInputStream()));
            PrintWriter pw=new PrintWriter(new OutputStreamWriter(socket.getOutputStream()));
            pw.print("HALLO\n");
            pw.flush();
            String mess=br.readLine();
            System.out.println("Message recu du serveur :"+mess);
            pw.close();
            br.close();
            socket.close();
        }
        catch(Exception e){
            System.out.println(e);
            e.printStackTrace();
        }
    }
}
```

Les informations liées à une socket

```
import java.net.*;
import java.io.*;
public class ClientEchoPort{
    public static void main(String[] args){
        try{
            Socket socket=new Socket("monjetas",7);
            System.out.println("La socket est connectee");
            System.out.println("Le port local est "+socket.getLocalPort());
            System.out.println("Le port distant est "+socket.getPort());
            System.out.println("La machine locale est "+socket.getLocalAddress().getHostName());
            System.out.println("La machine distante est "+socket.getInetAddress().getHostName());
            socket.close();
        }
        catch(Exception e){
            System.out.println(e);
            e.printStackTrace();
        }
    }
}
```