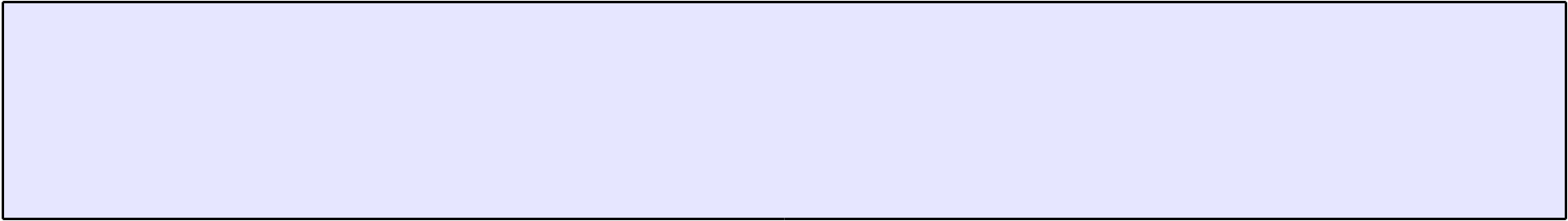
Flush et close

* 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

PR - API TCP Java 10

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));**

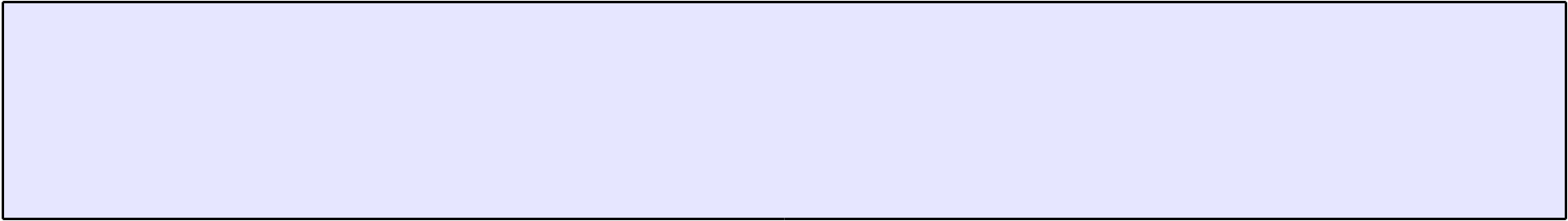
**String s=bf.readLine()**

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

PR - API TCP Java 18

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

PR - API TCP Java 21

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**)

PR - API TCP Java 24

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);**

PR - API TCP Java 26

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

PR - API TCP Java 27

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();**

**}**

PR - API TCP Java 28

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**

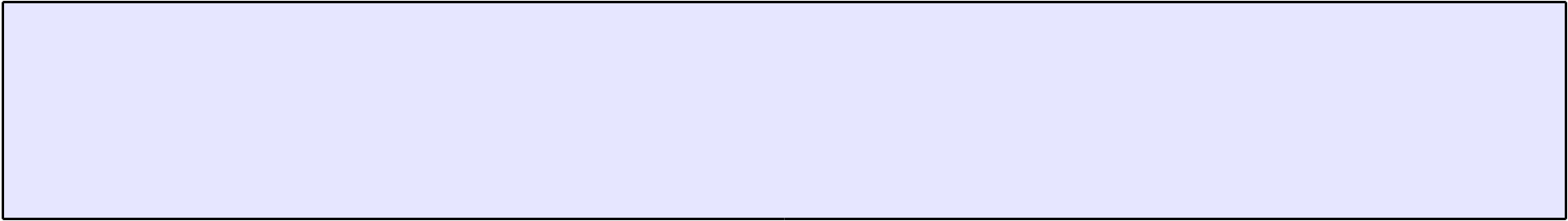
**IOException**

* + - 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**

PR - API TCP Java 32

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



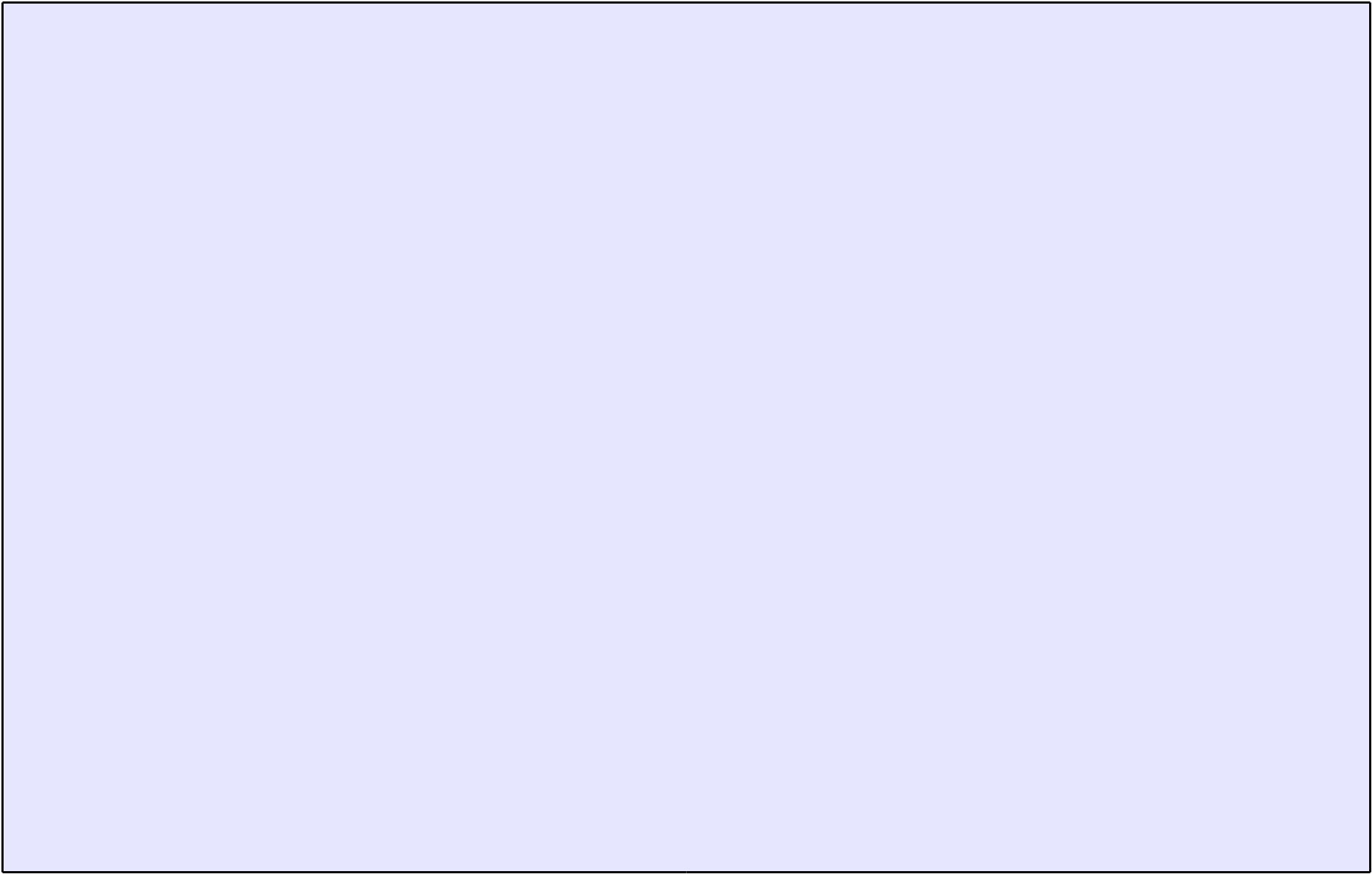
**Socket socket=new Socket("lampe",7); BufferedReader br=new BufferedReader(**

**new InputStreamReader(socket.getInputStream())); PrintWriter pw=new PrintWriter(**

**new OutputStreamWriter(socket.getOutputStream()));**

PR - API TCP Java 33

Création d'un client TCP



**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();**

**}**

**}**

**}**

35

Création d’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

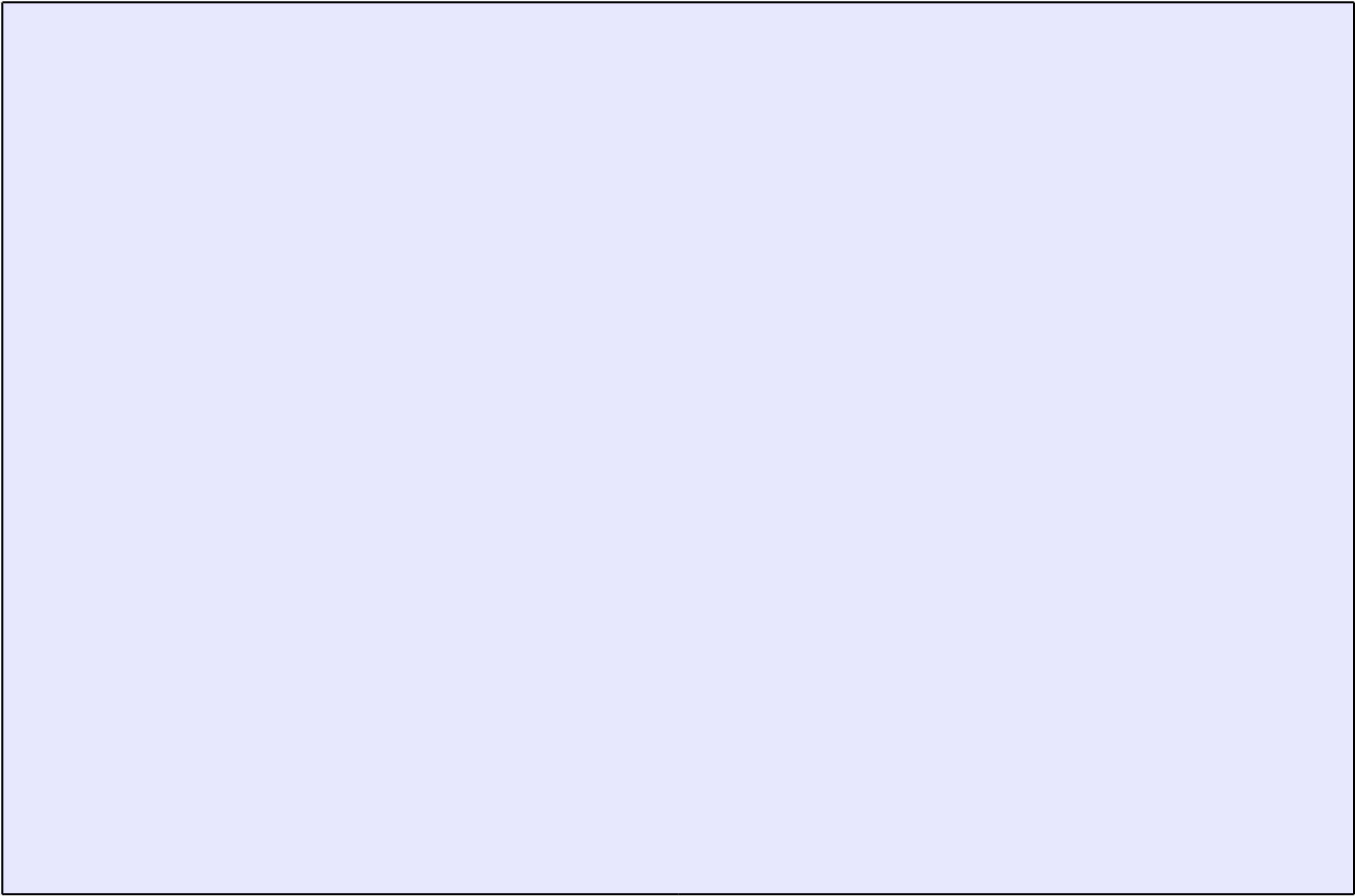
PR - API TCP Java 38

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)

PR - API TCP Java 40

Création d'un serveur TCP



**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();**

**}**

**}**

**}**

Pour programmer notre client

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

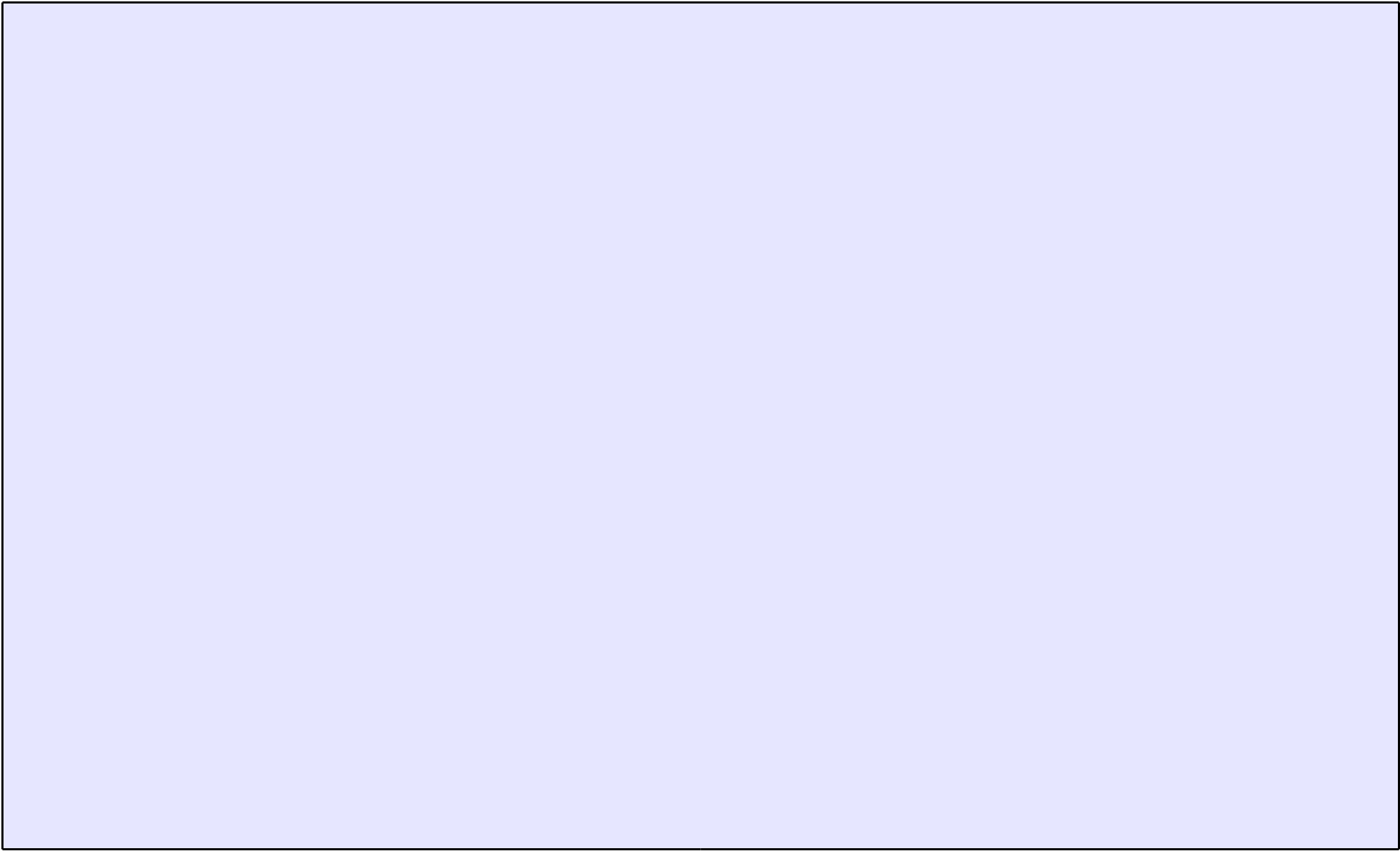


**Socket socket=new Socket("lulu",4242);**

* 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

PR - API TCP Java 45

Client TCP pour notre serveur



**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);**

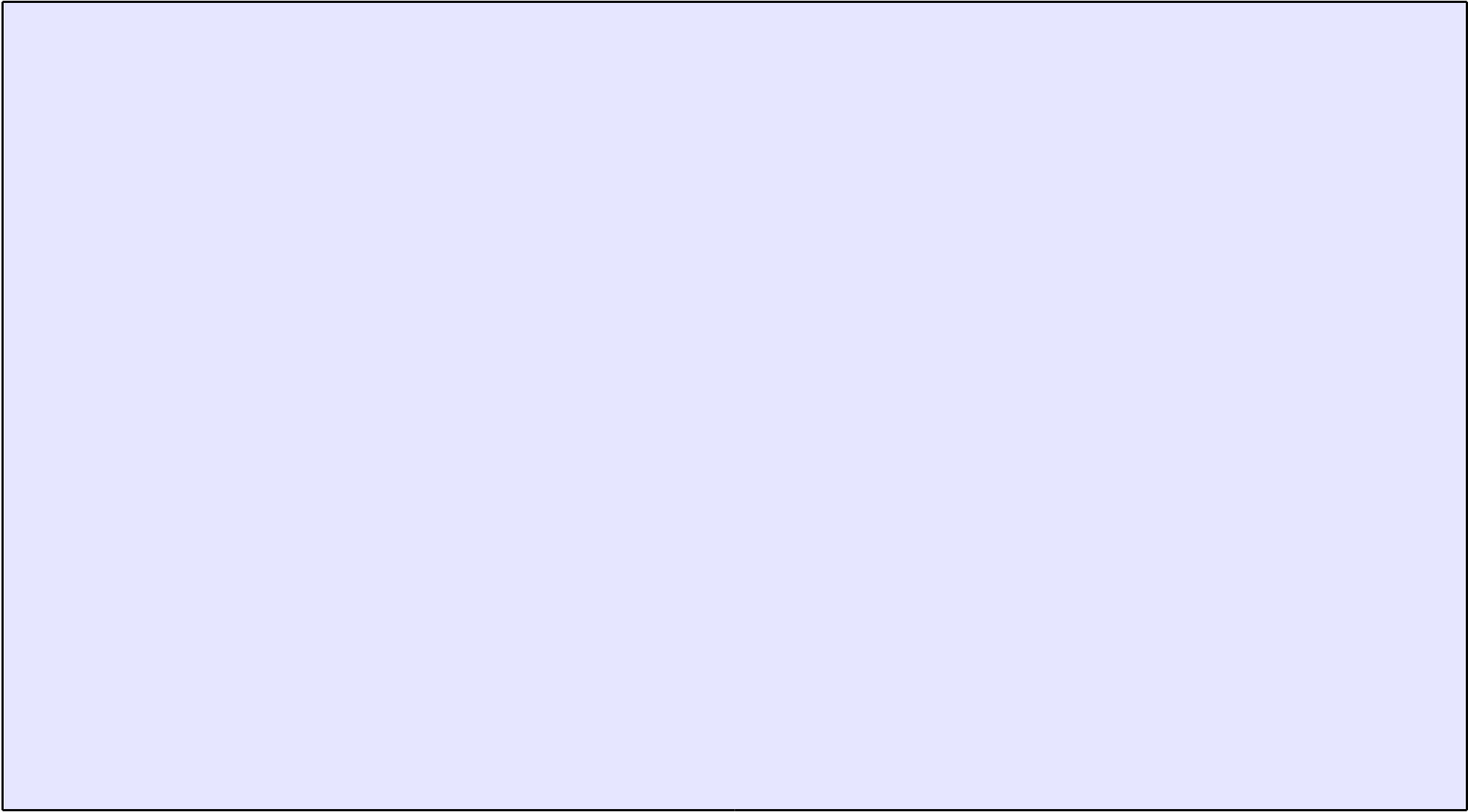
**e.printStackTrace();**

**}**

**}**

**}**

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();**

**}**

**}**

**}**

PR - API TCP Java 49