http://www.labo-sun.com
labo-sun@supinfo.com

Les Sockets

COMMUNICATION RESEAU

Auteur : Youssef Medaghri-alaoui
Version n° 2.0 – 14 octobre 2004
Nombre de pages : 29

Ecole Supérieure d'Informatique de Paris
23. rue Château Landon 75010 – PARIS
www.supinfo.com

Table des matières

1. INTRODUCTION

Le but de ce cours est de vous permettre d'apprendre à utiliser le package **java.net** contenant les éléments nécessaires à la programmation réseau.

Les packages **java.net** et **javax.net** ainsi que les classes qu'ils fournissent font de Java un très bon langage de programmation réseau.

Les classes du package **java.net** encapsulent le concept des « **sockets** » du projet Berkeley Software Distribution (BSD), inventé à l'université de Berkeley en Californie.

1.1.Bases de la programmation réseau

En 1978, à Berkeley, un dénommé Bill Joy était à la tête d'un projet visant à apporter des évolutions importantes au système UNIX comme la mémoire virtuelle ou des fonctionnalités d'affichage en plein écran.

En 1984, Bill Joy décide de quitter ce projet pour fonder Sun Microsystems, et lance alors une version 4.2BSD d'UNIX appelée aussi le Berkeley UNIX et comprenant déjà des fonctionnalités de communication réseau.

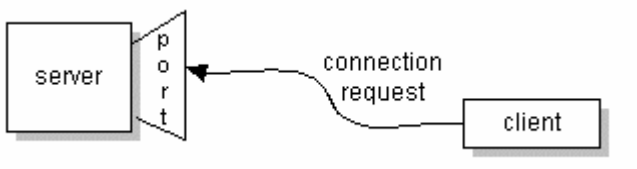
Le support de communication livré avec cette version 4.2 est finalement devenu un standard pour Internet. Rapidement les sockets ont été adoptées pour les communications réseaux et interprocessus, puis le concept a été intégré à la fin des années 80 à Windows et Macintosh.

1.2.Sockets

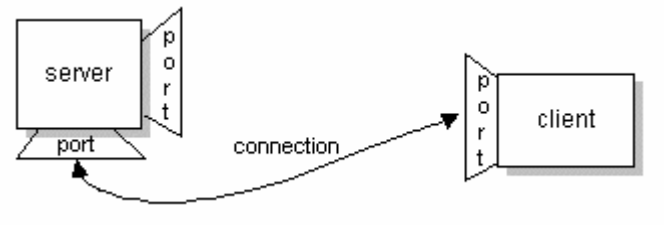
Un **socket** est un point de terminaison dans une communication bidirectionnelle entre deux programmes fonctionnant sur un réseau.

Un **socket** est associé à un numéro de port afin que la couche TCP puisse identifier l'application vers laquelle les données doivent être transmises.

En fonctionnement normal, une application serveur fonctionne sur un ordinateur et possède un **socket** d'écoute associé à un port d'écoute. Le serveur attend une demande de connexion de la part d'un client sur ce port.



Si tout se passe bien, le serveur accepte la connexion. Suite à cette acceptation, le serveur crée un nouveau **socket** associé à un nouveau port. Ainsi il pourra communiquer avec le client, tout en continuant l'écoute sur le socket initial en vue d'autres connexions.



2. Programmation Réseau en Java

2.1.java.net et javax.net

Les packages **java.net** et **javax.net** fournissent les classes et interfaces suivantes :

- Adresses IP : **InetAddress**
- Socket TCP : **Socket**, **ServerSocket**, **JSSE** (Java Secure Socket Layer)
- Socket UDP : **DatagramSocket**, **DatagramPacket**
- Socket MultiCast : **MulticastSocket**, **DatagramPacket**
- Classes niveau application (Couche 7 OSI) : **URL**, **URLConnection**, **HttpURLConnection**, **JarURLConnection**

Dans ce cours nous ne nous intéresserons pas à **JSSE**, aux **MulticastSocket**, et aux **URL**.

2.1.1. java.net.InetAddress

La classe **InetAddress** représente une adresse du protocole IP, aussi bien par son adresse IP que par son nom d'hôte.

Cette classe peut être vue comme la représentation d'une adresse IP est utilisée par les classes **Socket** et **DatagramSocket**.

Les trois méthodes suivantes permettent la résolution DNS :

Obtenir l'adresse ou les adresses de l'hôte dont le nom est passé en paramètre :

```
public static InetAddress getByName(String hostname) throws UnknownHostException
```

```
public static InetAddress[] getAllByName(String hostname) throws
UnknownHostException
```

Obtenir l'adresse en local:

```
public static InetAddress getLocalHost() throws UnknownHostException
```

2.1.2. Socket en mode connecté (TCP)

Les **sockets** TCP/IP sont utilisés pour établir des échanges de flux fiables, bidirectionnels et point à point entre différentes machines sur Internet.

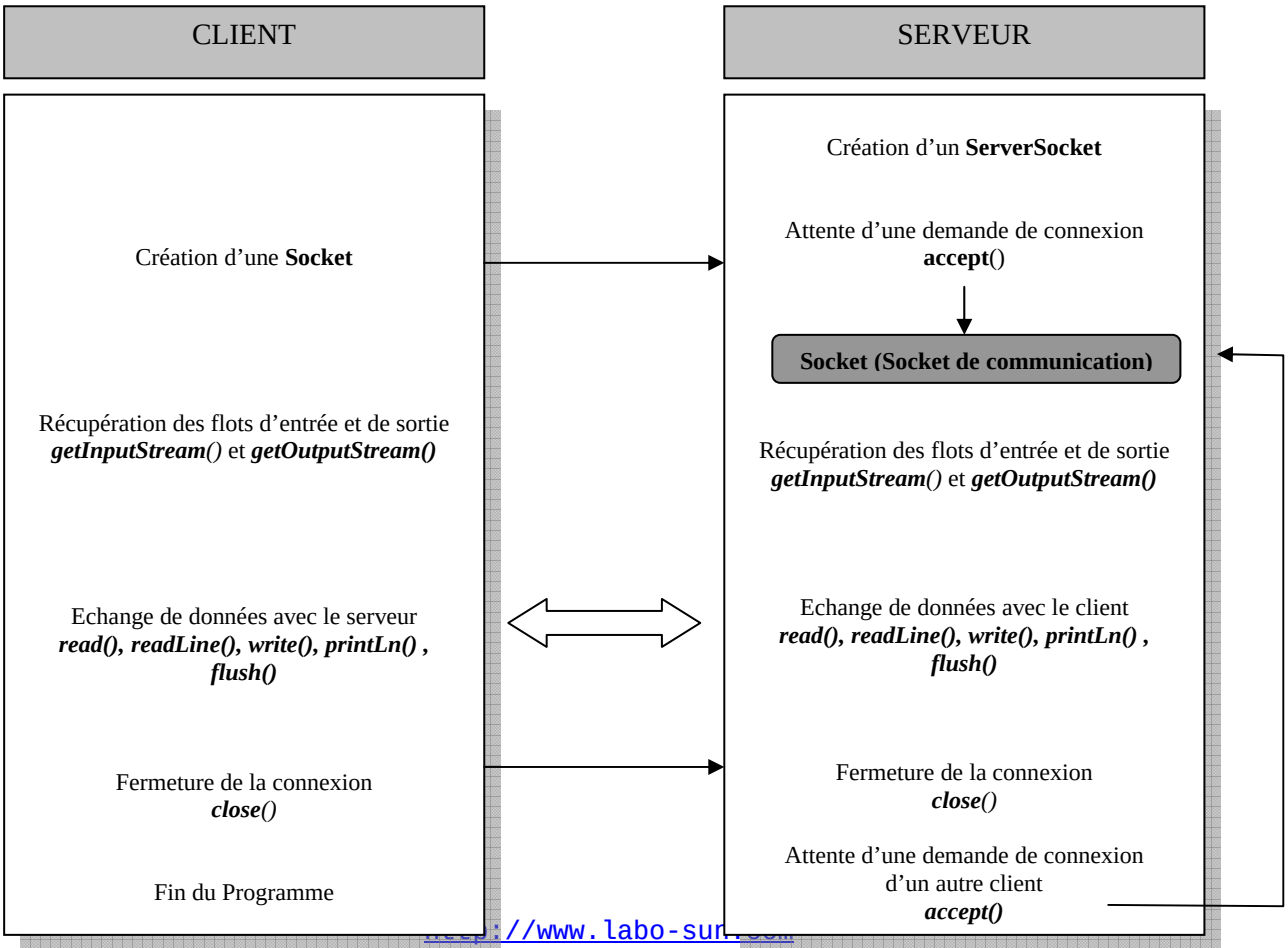
Les **sockets** peuvent également permettre à une application Java de communiquer avec une tout autre application de la machine locale où n'importe où sur Internet.

En mode connecté (TCP), une connexion fiable est établie entre client et serveur (contrôle d'erreur, renvoi de paquets…).

Java considère deux types de **sockets** TCP :

- La classe **ServerSocket** permet d'implémenter le serveur. Celui-ci attend les demandes de connexion et les accepte.
- La classe **Socket** permet d'implémenter un client. Il demande l'établissement de la connexion avec le serveur. C'est au sein de cette classe que l'on trouve les méthodes de communication.

Le processus d'exécution d'une communication par **socket** en mode connecté est présenté ci-dessous.



2.1.2.1. Serveur (java.net.ServerSocket)

Utilisation :

Création d'un **socket** d'écoute :

```
ServerSocket Serveur = new ServerSocket(port);
```

Attente des demandes et création du **Socket** de communication

```
Socket Communication = Serveur.accept();      // bloquant en attente de connexion
```

La communication se fera par le biais de la classe **Socket**

Constructeurs :

```
public ServerSocket(int port) throws IOException    // Crée une socket à l'écoute
du port spécifié.
```

```
public ServerSocket(int port, int backlog) throws IOException

/* Crée une socket à l'écoute du port spécifié, en spécifiant la taille de la file
d'attente des demandes de connexion. */
```

```
public ServerSocket(int port, int backlog, InetAddress bindAddr) throws IOException

/* Crée une socket à l'écoute du port spécifié, en spécifiant la taille de la file
d'attente des demandes de connexion, et en restreignant l'adresse sur laquelle on
accepte les connexions. */
```

Méthodes utiles :

```
public Socket accept() throws IOException /* Accepte la connexion et renvoie un
nouveau socket */
```

La méthode ci-dessus est bloquante, mais l'attente peut être limitée dans le temps grâce à la méthode :

```
public void setSoTimeout(int timeout) throws SocketException

/* Cette méthode prend en paramètre le délai de garde exprimé en millisecondes. La
valeur par défaut 0 équivaut à l'infini. À l'expiration du délai de garde,
l'exception java.io.InterruptedIOException est levée. */
```

```
public void close()       // Ferme le socket d'écoute.
```

```
public InetAddress getInetAddress()    // Retourne l'adresse à partir de laquelle
la socket écoute
```

```
public int getLocalPort()       // Retourne le port sur lequel la socket écoute
```

2.1.3. Client (java.net.Socket)

Utilisation :

Création de la **socket** cliente :

```
Socket Client = new Socket(host, port);
```

Communication : On utilisera les flux **InputStream**, **OutputStream** et leurs dérivés

```
InputStream entree = Client.getInputStream();

OutputStream sortie = Client.getOutputStream();
```

Constructeurs :

```
public Socket()           // Crée une socket non connectée
```

```
public Socket(InetAddress address, int port) throws IOException

/* Crée une socket de communication et établit une connexion sur le port voulu de
la machine dont l'adresse IP est spécifiée. */
```

```
public Socket (InetAddress ad, int port, InetAddress localAd, int localPort) throws
IOException

/* Crée une socket de communication et établit une connexion sur le port voulu de
la machine dont l'adresse IP est spécifiée. Spécifie en plus le port et l'adresse
locale. */
```

```
public Socket(String host, int port) throws UnknownHostException, IOException

/* Crée une socket de communication et établit une connexion sur le port voulu de
la machine dont le nom d'hôte est spécifié. */
```

```
public Socket(String host, int port, InetAddress localAd, int localPort) throws
IOException, UnknownHostException

/* Crée une socket de communication et établit une connexion sur le port voulu de
la machine dont le nom d'hôte est spécifié. Spécifie en plus le port et l'adresse
locale. */
```

Méthodes utiles :

1. Communication

La communication effective sur une connexion par **socket** est basée sur les flux de données (**java.io.OutputStream** et **java.io.InputStream**).

Les deux méthodes suivantes sont utilisées pour obtenir les flux en entrée et en sortie.

```
public InputStream getInputStream() throws IOException

// Retourne l'objet correspondant au flux d'entrée de l'objet socket
```

```
public OutputStream getOutputStream() throws IOException

// Retourne l'objet correspondant au flux de sortie de l'objet socket
```

2. Lecture bloquante

Une opération de lecture sur ces flots est bloquante tant que des données ne sont pas disponibles. Cependant, il est possible de fixer un délai de garde pour l'attente des données (similaire au délai de garde de la **socket** d'écoute : levée de l'exception **java.io.InterruptedIOException**).

```
public void setSoTimeout(int timeout) throws SocketException
```

3. Méthodes complémentaires

Un ensemble de méthodes permet d'obtenir les éléments constitutifs de la liaison établie ;

```
InetAddress getInetAddress()

/*
Retourne l'adresse à laquelle la socket est connectée
L'adresse de type InetAddress concatène l'adresse IP et le nom de domaine
correspondant
Par ex :  www.yahoo.com/216.109.117.207
*/
```

```
int getPort()      // Retourne le port distant sur lequel la socket est connectée
```

```
InetAddress getLocalAddress()    // Retourne l'adresse locale à laquelle la socket
est associée
```

```
int getLocalPort()        // Retourne le port local associé à la socket
```

Fermeture

L'opération close ferme la connexion et libère les ressources du système associées au **socket**.

```
void close()        // Ferme le socket
```

2.1.4. Un exemple de client et de serveur (très basique)

Voici un exemple de serveur qui accepte une connexion.

Il reçoit un entier puis renvoi au client (entier + 1).

Serveur :

```
import java.net.*;
import java.io.*;

public class ServeurBasique {

	public static void main(String[] args) {

		try {
			ServerSocket ecoute = new ServerSocket(18000, 5);
			Socket service = (Socket) null;

			while (true) {
				service = ecoute.accept();
				OutputStream os = service.getOutputStream();
				InputStream is = service.getInputStream();
				os.write(is.read() + 1);
				service.close();
			}
		} catch (Exception e) {
			/* traitement d'erreur */
		}
	}
}
```

Le client correspondant demande une connexion au serveur.

Le client envoie un entier, puis lit le flux d'entree de la socket.

Client :

```
import java.net.*;
import java.io.*;

public class ClientBasique {

	public static void main(String[] args) {

		try {
			Socket s = new Socket("localhost", 18000);
			OutputStream os = s.getOutputStream();
			InputStream is = s.getInputStream();
			os.write((int) 'a');
			System.out.println((char) is.read());
			s.close();
		} catch (Exception e) {
			/* Traitement d'erreur */
		}
	}
}
```

2.1.5. Compréhension des exceptions

Dans certains cas, par exemple si on a une demande de connexion à un serveur qui n'a pas de **socket** d'écoute en attente, une exception est levée.

La liste ci-dessous vous permettra de comprendre la signification des exceptions levées en cas de problème.

```
java.net.BindException    // Erreur de liaison à une adresse locale liée
```

```
java.net.ConnectException        // Refus de connexion par l'hôte
```

```
java.net.NoRouteToHostException  // Hôte injoignable (route non trouvée)
```

```
java.net.ProtocolException       // Erreur de protocole (TCP, ...)
```

```
java.net.SocketException         // Erreur de protocole (TCP, ...)
```

```
java.net.UnknownHostException    // Erreur de DNS
```

2.1.6. Sockets en mode datagramme

Le protocole TCP/IP répond à la majorité des besoins réseaux. Il permet de transporter de manière très fiable des paquets de données. Toutefois, du fait de l'exécution de nombreux algorithmes de traitement des congestions du réseau ou des paquets perdus, il constitue une solution de transport coûteuse.

Les datagrammes représentent une solution alternative.

Un datagramme est un ensemble d'informations transmissibles de machine en machine. Aucun contrôle n'existe pour déterminer si un datagramme est bien arrivé à destination ou s'il a été intercepté par une autre machine. De même, il n'existe aucun moyen permettant de savoir s'il a été endommagé au cours du transfert.

Java implémente les datagrammes dans le cadre du protocole UDP, et fournit deux classes :

- **DatagramPacket** : Permet de créer des objets contenant les données du datagramme. Un **DatagramPacket** possède une zone de données, un numéro de port et éventuellement une adresse IP.
- **DatagramSocket** : Mécanisme qui permet d'envoyer et de recevoir des **DatagramPacket**.

2.1.6.1. Constructeurs

```
public DatagramSocket() throws SocketException
// Construit un socket Datagramme.
```

```
public DatagramSocket(int port) throws SocketException
// Construit un socket Datagramme en spécifiant un port sur la machine locale.
```

```
public DatagramPacket(byte[] buf, int length)
// Construit un paquet en mode Datagramme.
```

```
public DatagramPacket(byte[] buf, int length, InetAddress address, int port)
// Construit un paquet en mode Datagramme en spécifiant l'adresse et le port de
destination.
```

2.1.6.2. Emission/Réception

```
public void send(DatagramPacket p) throws IOException      // Envoyer un paquet
```

```
public void receive(DatagramPacket p) throws IOException   // Recevoir un paquet
```

2.1.6.3. Connexion

Il est possible de « connecter » un **socket** en mode datagramme à un destinataire. Dans ce cas, les paquets émis sur le **socket** le seront toujours pour l'adresse spécifiée.

La connexion simplifie l'envoi d'une série de paquets (il n'est plus nécessaire de spécifier l'adresse de destination pour chacun d'entre eux) et accélère les contrôles de sécurité (ils ont lieu une fois pour toute à la connexion).

La « déconnexion » enlève l'association (le **socket** redevient disponible comme dans l'état initial).

```
public void connect(InetAddress address, int port)
```

```
public void disconnect()
```

2.1.6.4. Autres Méthodes

```
public InetAddress getAddress()
// Retourne l'adresse de destination d'un paquet sous la forme d'un objet
InetAddress.
```

```
public int getPort()
// Retourne le numéro du port de destination.
```

```
public byte[] getData()
// Retourne le tableau de données fourni par un datagramme. On utilise souvent
cette méthode à la réception d'un objet DatagrapPacket.
```

```
public int getLength()
// Retourne la taille des données valides contenues dans le tableau de données du
datagramme.
```

```
public void close()
// Libère les ressources du système associées à la socket.
```

Exemple :

Envoyer un message en mode datagramme :

```
import java.net.*;
import java.io.*;

public class EnvoiMessage {
	// Port sur lequel on envoie les données
	static final int port=47;

	// Point d'entrée du programme
	public static void main(String argv[]) {

		// On vérifie si l'adresse IP ou le nom de la machine destinataire est
spécifié.

		if (argv.length != 1) {
			System.err.println("Donnez le nom de la machine destinataire");
			System.exit(0);
		}

		BufferedReader entree = new BufferedReader(new
InputStreamReader(System.in));

		InetAddress adresse = null;
		DatagramSocket socket;

		// Création de l'adresse du destinataire a partir de l'argument
		try {
			adresse = InetAddress.getByName(argv[0]);

			// Envoi du message
			String ligne = "Hello world";
			byte[] message = new byte[ligne.length()];

			message = ligne.getBytes();
			DatagramPacket envoi = new DatagramPacket(message,
ligne.length(), adresse,port);
			socket = new DatagramSocket();
			socket.send(envoi);
		} catch (Exception exc) {
			System.err.println("Erreur lors de l'envoi du message !");
		}
	}
}
```

Recevoir un message en mode datagramme :

```
import java.net.*;
import java.io.*;

class ReceptionMessage {
	// Port de réception
	static final int port=47;

	// Méthode principale
	public static void main(String argv[]) {
		byte[] receptionOctets = new byte[1000];
		String texte;

		try {
			DatagramSocket socket = new DatagramSocket(port);
			DatagramPacket reception = new DatagramPacket(receptionOctets,
receptionOctets.length);
			socket.receive(reception);
			texte = new String(receptionOctets);
			System.out.println("Reception de la machine " +
reception.getAddress().getHostName());
			System.out.println("Sur le port " + reception.getPort());
			System.out.println("Texte:" + texte);
		} catch (Exception e) {
			System.err.println("Erreur lors de l'ouverture du socket");
		}
	}
}
```

3. Serveur multi-threadé

3.1. Rappel sur les Threads

Java propose une prise en charge intégrée de la programmation multi-threadée grâce à la classe **Thread** et l'interface **Runnable**.

Un programme multi-threadé renferme plusieurs parties susceptibles de fonctionner en parallèle.

Chacune de ces parties est appelée thread. Les threads partagent le même espace d'adressage et exploitent en coopération le même processus système.

3.2. Classe Thread et interface Runnable

Le système de multithreading de Java repose sur la classe **Thread,** ses méthodes, et sur son interface associée **Runnable**.

Pour créer un nouveau thread le programme principal peut soit étendre la classe **Thread**, soit mettre en œuvre l'interface **Runnable**.

```
public class MonProgramme extends Thread { ...

public class MonProgramme implements Runnable { ...
```

Plusieurs méthodes de la classe **Thread** facilitent la gestion des threads :

- ***getName()*** : Obtenir le nom d'un thread
- ***getPriority()*** : Obtenir la priorité d'un thread
- ***isAlive()*** : Déterminer si un thread est toujours actif
- ***join()*** : Attendre la fin d'un thread
- ***run()*** : Point d'entrée pour un thread
- ***sleep()*** : Mettre un thread en attente pendant une certaine durée
- ***start()*** : Démarrer un thread

3.2.1. Thread principal

L'exécution d'un programme Java implique obligatoirement la création d'un thread. Le thread principal, à partir duquel peuvent être générés d'autres threads, appelés les threads enfants.

Bien que créé automatiquement à l'exécution d'un programme, il peut être lui aussi contrôlé par un objet **Thread**.

La méthode ***currentThread()***, permet d'obtenir une référence du thread dans lequel elle s'exécute.

Il est ensuite possible de le contrôler comme n'importe quel autre thread.

```
public class MonProgramme {

	public static void main(String args[]) {
		Thread t = Thread.currentThread();
		System.out.println("Thread : " + t + " en cours");
		t.setName("Mon Thread");  // Changement de nom

		try {
			t.sleep(2000);
		} catch(InterruptedException e) { }

		System.out.println("Reprise du Thread : " + t + " et fin du programme
");
	}
}
```

3.2.2. Créer un thread

La manière la plus simple de créer un thread consiste à créer une classe qui implémente l'interface **Runnable**.

La deuxième manière de procéder consiste à créer une classe qui étende la classe **Thread**.

3.2.2.1. Utiliser Runnable

Pour pouvoir implémenter l'interface **Runnable**, une classe doit posséder la méthode suivante :

```
public void run()
```

Cette méthode constitue le point d'entrée d'un nouveau thread concurrent au sein du programme.

Un thread se termine à la sortie de ***run()***.

L'instanciation d'un objet de type **Thread** se fait à l'aide du constructeur suivant :

```
Thread(Runnable ObThread, String NomThread)
```

ObThread désigne une instance d'une classe qui met en œuvre l'interface **Runnable**. *NomThread* désigne le nom que l'on veut lui donner. Le nouveau thread ne devient actif qu'après l'appel de sa méthode ***start()***.

Exemple de programmation multithreadée grâce à **Runnable**.

```
public class MonProgramme {

	public static void main(String args[]) {

		new NouveauThread("Enfant 1");
		new NouveauThread("Enfant 2");

		Thread t = Thread.currentThread();
		t.setName("Parent");

		for (int i = 0; i < 10; i++) {
			System.out.println("Thread : " + t.getName() + " en cours");
		}
	}
}

class NouveauThread implements Runnable {

	Thread t;

	NouveauThread(String nom) {
		t = new Thread(this, nom);
		t.start();
	}

	public void run() {
		for (int i = 0; i < 10; i++) {
			System.out.println("Thread : " + t.getName() + " en cours");
		}
	}
}
```

3.2.2.2. Etendre Thread

Il s'agit ici de définir une nouvelle classe qui étende **Thread**.

La classe étendue doit redéfinir la méthode ***run()***, qui est ici aussi le point d'entrée du nouveau thread. Elle doit également invoquer ***start()*** afin de lancer l'exécution du thread.

On peut se demander pourquoi Java propose deux manières de créer des threads enfants. En fait, la classe **Thread** possède plusieurs méthodes redéfinissables par une classe dérivée, dont ***run()***.

Toutefois dans le cas où l'on ne redéfinit que ***run()*** cela revient à utiliser **Runnable**.

Chacun est libre de choisir la méthode qui lui convient.

L'exemple précédent est repris en page suivante mais cette fois ci avec la méthode **extends Thread**.

Reprise de l'exemple précédent en héritant de la classe **Thread** :

```
public class MonProgramme {

	public static void main(String args[]) {

		new NouveauThread("Enfant 1");
		new NouveauThread("Enfant 2");

		Thread t = Thread.currentThread();
		t.setName("Parent");

		for (int i = 0; i < 10; i++) {
			System.out.println("Thread : " + t.getName() + " en cours");
		}
	}
}

class NouveauThread extends Thread {

	NouveauThread(String Nom) {
		setName(Nom);
		start();
	}

	public void run() {
		for (int i = 0; i < 10; i++) {
			System.out.println("Thread : " + getName() + " en cours");
		}
	}
}
```

3.3.Serveur multithreadé et client(s) simple(s)

Voici un exemple de serveur multithreadé basique qui accepte des connexions, attend une ligne de texte ("Bonjour") puis répond ("Ravi de vous entendre") :

```
import java.net.*;
import java.io.*;

public class Serveur {

	public static void main(String [] args) {

		try {
			ServerSocket ecoute = new ServerSocket(18000, 5);
```

```
                    while (true){

                          new NouvelleConnexion(ecoute.accept());
                    }
              } catch (Exception e) {
                    /* traitement d'erreur */
              }
       }
}

class NouvelleConnexion implements Runnable {

       Thread t;
       Socket sck;
       PrintWriter sortie;
       BufferedReader entree;

       NouvelleConnexion(Socket sck) {
              t = new Thread(this, sck.getInetAddress().getHostName() + "/" +
sck.getPort());
              this.sck = sck;
              try {
                    sortie = new PrintWriter(this.sck.getOutputStream());
                    entrée = new BufferedReader(new
InputStreamReader(this.sck.getInputStream()));
              } catch (IOException e) {
              }
              t.start();
       }

       public void run() {
              System.out.println("Nouvelle Connexion acceptee : " + t.getName());
              try {
                    System.out.println("Recu : " + entree.readLine() + " sur " +
t.getName());
                    sortie.println("Ravi de vous entendre");
                    sortie.flush();
                    sck.close();
              } catch (IOException e) {
              }
       }
}
```

Code du Client :

```
import java.net.*;
import java.io.*;

public class Client {

       public static void main(String [] args) {

              try {
                    Socket connexion = new Socket("localhost", 18000);

                    PrintWriter sortie = new
PrintWriter(connexion.getOutputStream());
                    BufferedReader entrée = new BufferedReader(new
InputStreamReader(connexion.getInputStream()));

                    sortie.println("Bonjour");
                    sortie.flush();

                    System.out.println(entree.readLine());
                    connexion.close();
```

```
            } catch (Exception e) {
                  /* traitement d'erreur */
            }
      }
}
```

4. La Communication à l'aide des flux

Pour communiquer en réseau via TCP/IP, c'est la classe **Socket** qui fournit l'accès aux flux d'envoi et de réception.

Avant la version 1.1 du JDK, les classes d'entrées/sorties ne supportaient que des flux d'un octet. Le concept de flux de caractères, c'est à dire supportant des caractères Unicode sur 16 bits a été introduit en même temps que le JDK 1.1.

Il reste qu'au plus bas niveau, tout le système d'entrées/sorties est orienté octet. Les flux de caractères quant à eux constituent un outil pratique et efficace pour la gestion des caractères.

4.1. Les flux d'octets, InputStream et OutputStream

Les flux d'octets sont définis au moyen de deux hiérarchies de classes, à la tête desquelles figurent les classes abstraites **InputStream** et **OutputStream**.

Chacune d'elles possède plusieurs sous classes concrètes qui gèrent les différences entre divers dispositifs, comme des fichiers stockés sur un disque, des zones de mémoires tampons…

Les classes abstraites **InputStream** et **OutputStream** définissent plusieurs méthodes essentielles, mises en œuvre par les autres classes de flux.

Parmi les plus importantes, on peut citer ***read()*** et ***write()***, qui lisent et écrivent respectivement des flux de données.

Ces deux méthodes déclarées comme abstract sont redéfinies par chacune des classes dérivées.

Les flux **InputStream** et **OutputStream** peuvent être directement obtenus à partir de la socket à l'aide des méthodes suivantes :

```
Socket client = new Socket("ftp.free.fr", 21);

InputStream FluxReception = client.getInputStream();

OutputStream FluxEnvoi = client.getOutputStream();
```

4.2. Les flux de caractères

Les flux de caractères sont définis au moyen de deux hiérarchies de classes, à la tête desquelles figurent deux classes abstraites, **Reader** et **Writer**.

Celles-ci gèrent les flux de caractères Unicode. Java inclut plusieurs sous-classes concrètes de chacune d'elles.

Les classes abstraites **Reader** et **Writer** définissent plusieurs méthodes essentielles pour les autres classes de flux. Parmi les plus importantes, on trouve ***read()*** et ***write()*** qui lisent et écrivent respectivement des caractères de données. Elles sont redéfinies par chacune des classes de flux dérivées de **Reader** et **Writer**.

En programmation réseau, lorsque l'on voudra passer de ***getInputStream()*** ou ***getOutputStream()***, à un flux de caractères, il pourra être nécessaire de convertir au préalable les octets en caractères (dans le cas de **InputStream**) ou les caractères en octets (dans le cas de **OutputStream**).

```
Socket client = new Socket("ftp.free.fr", 21);

BufferedReader FluxReception = new BufferedReader(
      new InputStreamReader(client.getInputStream())
);

BufferedWriter FluxEnvoi = new BufferedWriter(
      new OutputStreamWriter(client.getOutputStream())
);
```

Cela ne sera pas systématiquement le cas, par exemple pour **PrintWriter** on aura

```
PrintWriter FluxEnvoi = new PrintWriter(client.getOutputStream());
```

Il convient donc de se référer à la documentation Java dès que cela s'avère nécessaire.

4.3.Classes d'entrées/sorties de flux d'octets

Les classes d'E/S des flux d'octets sont brièvement présentées ci-dessous :

- **BufferedInputStream** : Flux d'entrée stocké dans une zone mémoire tampon
- **BufferedOutputStream** : Flux de sortie stocké dans une zone mémoire tampon

- **ByteArrayInputStream** : Flux d'entrée lisant à partir d'un tableau d'octets
- **ByteArrayOutputStream** : Flux de sortie écrivant vers un tableau d'octets

- **DataInputStream** : Flux d'entrée qui renferme des méthodes dédiées à la lecture des types de données standard de Java
- **DataOutputStream** : Flux de sortie qui renferme des méthodes dédiées à l'écriture des types de données standard de Java

- **FileInputStream** : Flux d'entrée lisant à partir d'un fichier
- **FileOutputStream** : Flux de sortie écrivant vers un fichier

- **FilterInputStream** : Met en œuvre **InputStream** et surcharge ses méthodes
- **FilterOutputStream** : Met en œuvre **OutputStream** et surcharge ses méthodes

- **InputStream** : Classe abstraite décrivant le flux d'entrée
- **OutputStream** : Classe abstraite décrivant le flux de sortie

- **PipedInputStream** : Tube d'entrée
- **PipedOutputStream** : Tube de sortie

- **PrintStream** : Flux de sortie renfermant ***print()*** et ***println()***

- **PushBackInputStream** : Flux d'entrée qui permet de reculer d'un octet

- **RandomAccessFile** : Prend en charge l'E/S de fichier avec accès aléatoire

- **SequenceInputStream** : Flux d'entrée, composé de plusieurs flux d'entrée, qui seront lus de manière séquentielle, l'un après l'autre

4.4.Classes d'Entrées/Sorties des flux de caractères

Les classes d'E/S des flux de caractères sont brièvement présentées ci-dessous :

- **BufferedReader** : Flux de caractères d'entrée, stockés dans une zone mémoire tampon
- **BufferedWriter** : Flux de caractères de sortie, stockés dans une zone mémoire Tampon

- **CharArrayReader** : Flux d'entrée lisant à partir d'un tableau de caractères
- **CharArrayWriter** : Flux de sortie écrivant vers un tableau de caractères

- **FileReader** : Flux d'entrée lisant à partir d'un fichier
- **FileWriter** : Flux de sortie écrivant vers un fichier

- **InputStreamReader** : Flux d'entrée transcrivant des octets en caractères
- **OutputStreamReader** : Flux de sortie transcrivant des caractères en octets
- **LineNumberReader** : Flux d'entrée comptant des lignes

- **PipedReader** : Tube d'entrée
- **PipedWriter** : Tube de sortie

- **PrintWriter** : Flux de sortie renfermant ***print()*** et ***println()***

- **PushBackReader** : Flux d'entrée qui permet de retourner des caractères au flux d'entrée

- **Reader** : Classe abstraite, décrivant l'entrée de flux de caractères
- **Writer** : Classe abstraite, décrivant la sortie de flux de caractères

- **StringReader** : Flux d'entrée lisant à partir d'une chaîne
- **StringWriter** : Flux de sortie écrivant vers une chaîne

5. Transmission d'un objet sérialisé

5.1. Envoi d'un objet sérialisé

La sérialisation permet de sauvegarder l'état interne d'un objet, c'est-à-dire ses variables, puis de les le récupérer dans un flux. On peut donc envoyer les informations de cette instance par réseau après avoir utilisé la sérialisation.

Tout d'abord, l'objet que l'on souhaite sérialiser doit implémenter l'interface **Serializable**.

Ensuite on instancie un **Socket** en spécifiant par exemple l'adresse IP et le port, ou en acceptant une connexion entrante.

Puis on déclare un **BufferedOutputStream** qui prend le flux du **Socket** en paramètre (***Socket.getOutputStream()***).

La sérialisation d'un objet est effectuée lors de l'appel de la méthode ***writeObject()*** sur un objet implémentant l'interface **ObjectOutput** (par exemple un objet de la classe **ObjectOutputStream**) en passant en paramètre de la méthode ***writeObject()*** l'objet que l'on veut sérialiser, et en paramètre du constructeur **ObjectOutputStream** le flux sur lequel on envoie l'objet sérialisé.

5.2. Réception d'un objet sérialisé

La réception de l'objet sérialisé fonctionne quasiment de la même façon que pour l'émission.

On initialise un **Socket**. On déclare un **BufferedInputStream** qui prend en paramètre le flux entrant du socket.

La désérialisation d'un objet est effectuée en appelant la méthode ***readObject()*** sur un objet implémentant l'interface **ObjectInput** (par exemple un objet de la classe **ObjectInputStream**).

5.3. Exemples

Notre exemple se divise en trois classes : une classe **Personne** dont une instance sera sérialisée et envoyée par le réseau. La classe **Envoi** se charge d'envoyer cette instance de **Personne**, et la classe **Reception** recevra l'instance et affichera la valeur des variables d'instance pour vérifier que tout c'est bien déroulé.

5.3.1. Classe Personne

```
import java.io.*;

public class Personne implements Serializable {
	String nom;
	int age;

	public Personne(String n, int a) {
		nom = n;
		age = a;
	}

	public void affiche() {
		System.out.println("nom:" + nom);
```

```
                System.out.println("age:" + age);
        }
}
```

5.3.2. Classe Envoi

```
import java.io.*;
import java.net.*;

public class Envoi {

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

                Socket client = new Socket("localhost", 21);

                BufferedOutputStream out = new
BufferedOutputStream(client.getOutputStream());

                ObjectOutput s = new ObjectOutputStream(out);
                s.writeObject(new Personne("Stephane", 22));

                s.flush();
        }
}
```

5.3.3. Classe Reception

```
import java.io.*;
import java.net.*;

public class Reception implements Serializable{


        public Reception() {
        }

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

                ServerSocket serveur = new ServerSocket(21);
                Socket communication = serveur.accept();
                // bloquant en attente de connexion

                BufferedInputStream in = new
BufferedInputStream(communication.getInputStream());

                ObjectInput s = new ObjectInputStream(in);
                Personne e = (Personne) s.readObject();

                e.affiche();
        }
}
```

6. Programmation des Entrées/Sorties

Pour programmer des entrées/sorties sur le réseau, on utilise principalement l'une des quatre classes suivantes, mais libre à vous d'en choisir d'autres (**ZipOutputStream**....)

- **java.io.BufferedReader** : Cette classe est utilisée pour une lecture avec mise en mémoire tampon, et non continue, des données entrantes d'un **Socket**.
- **java.io.PrintWriter** : On utilise cette classe lorsque l'on traite des protocoles basés sur des requêtes texte (***print()***, ***println()***)
- **java.io.BufferedInputStream** : Cette classe est parfaite lorsqu'on souhaite réaliser des transferts binaires et continus entre deux sockets. Elle permet la mise en mémoire tampon des données et accélère le processus de lecture.
- **java.io.BufferedOutputStream** : Equivalente à la précédente pour tout ce qui concerne les sorties d'un **Socket**.

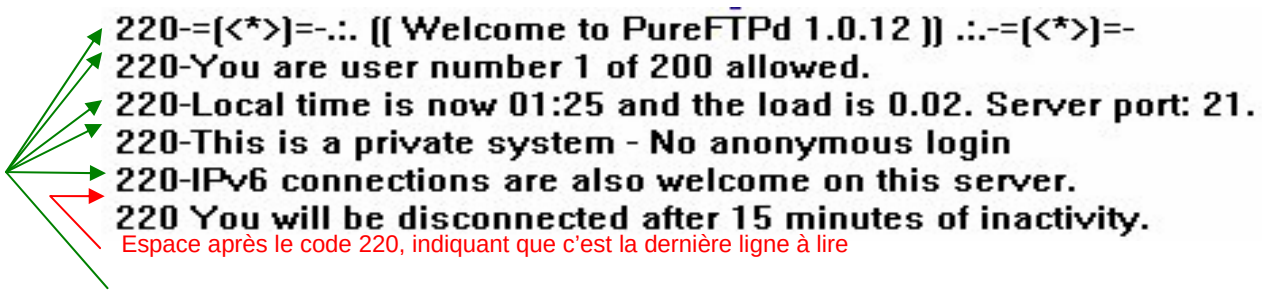
6.1. BufferedReader et PrintWriter

Ces classes sont très souvent utilisées de façon conjointe pour lire et écrire vers un **Socket**. Pour détecter la réception d'un *EOF*[1] (qui arrive quand le **Socket** distant est fermé), on teste la valeur **null**. Une fois les instances de ces classes créées, il est très simple d'organiser une communication basée sur un protocole. En effet ces classes fournissent les méthodes ***print()***, ***read()***, ***println()***, ***readLine()***.

1) Différents choix peuvent être opérés, comme par exemple l'envoi du nombre de lignes suivi de l'envoi de ces lignes, une boucle for peut alors être utilisée pour les lectures successives.

2) Une autre solution consiste à mettre en place un protocole basé sur une séquence de fin, comme par exemple l'envoi de "END" ou "FINI" à la fin d'une séquence d'envois.

3) Enfin on peut se pencher sur la méthode mise en place par le protocole ftp pour indiquer si il reste des lignes à lire.



[1] End Of File

On peut constater que lorsqu'il reste des lignes à lire on se trouve avec la séquence code+'-'.

Par contre lorsque la dernière ligne est envoyée, on trouve la séquence code+' '. A nous de mettre en place une détection de ces informations (à l'aide de ***substring()*** par ex), pour éviter de faire un ***readLine()*** de trop.

6.2.BufferedInputStream et BufferedInputStream

On utilise généralement ces classes pour des transferts binaires continus. L'envoi binaire à l'aide de ces classes est facile à mettre en œuvre. Etant donné que le transfert est binaire, le type le plus utilisé sera **int**.

Pour détecter la réception d'un *EOF* (qui arrive quand le **Socket** distant est fermé), on teste la valeur -1.

Voici un exemple de lecture binaire :

```
try {
	Socket socket = new Socket(host, port);// Formation de la socket
	BufferedInputStream in = new BufferedInputStream(socket.getInputStream());
	int ch;

	while ((ch = in.read()) != -1){
		/* Mise dans un Fichier, ... affichage sous forme de char ... */
	}

	in.close();
	socket.close();

} catch (Exception exception) {
	exception.printStackTrace();
}
```